2011

On the Factor Refinement Principle and its Implementation on Multicore Architectures

Md. Mohsin Ali

On the Factor Refinement Principle and its Implementation on Multicore Architectures

(Spine title: Factor Refinement Principle on Multicore Architectures)

(Thesis format: Monograph)

by

Md. Mohsin Ali

Graduate Program
in
Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO
THE SCHOOL OF GRADUATE AND POSTDOCTORAL STUDIES

CERTIFICATE OF EXAMINATION

Supervisor:	Examination committee:
Dr. Marc Moreno Maza	Dr. Éric Schost
	Dr. Sheng Yu
	Dr. Lex E. Renner

The thesis by

Md. Mohsin Ali

entitled:

On the Factor Refinement Principle and its Implementation on Multicore Architectures

is accepted in partial fulfillment of the
requirements for the degree of
Master of Science

Date _____ Chair of the Thesis Examination Board

Abstract

The *factor refinement principle* turns a partial factorization of integers (or polynomials) into a more complete factorization represented by basis elements and exponents, with basis elements that are pairwise coprime.

There are lots of applications of this refinement technique such as simplifying systems of polynomial inequations and, more generally, speeding up certain algebraic algorithms by eliminating redundant expressions that may occur during intermediate computations.

Successive GCD computations and divisions are used to accomplish this task until all the basis elements are pairwise coprime. Moreover, square-free factorization (which is the first step of many factorization algorithms) is used to remove the repeated patterns from each input element. Differentiation, division and GCD calculation operations are required to complete this pre-processing step. Both factor refinement and square-free factorization often rely on plain (quadratic) algorithms for multiplication but can be substantially improved with asymptotically fast multiplication on sufficiently large input.

In this work, we review the working principles and complexity estimates of the factor refinement, in case of plain arithmetic, as well as asymptotically fast arithmetic. Following this review process, we design, analyze and implement parallel adaptations of these factor refinement algorithms. We consider several algorithm optimization techniques such as data locality analysis, balancing subproblems, etc. to fully exploit modern multicore architectures. The `Cilk++` implementation of our parallel algorithm based on the augment refinement principle of Bach, Driscoll and Shallit achieves linear speedup for input data of sufficiently large size.

Keywords. Factor refinement, Coprime factorization, Square-free factorization, GCD-free basis, Parallelism, Muticore architectures.

Acknowledgments

Firstly, I would like to thank my thesis supervisor Dr. Marc Moreno Maza in the Department of Computer Science at The University of Western Ontario. His helping hands toward the completion of this research work were always extended for me. He consistently helped me on the way of this thesis and guided me in the right direction whenever he thought I needed it. I am grateful to him for his excellent support to me in all the steps of successful completion of this research.

Secondly, I would like to thank Dr. Yuzhen Xie in the Department of Computer Science at The University of Western Ontario for helping me successfully completing this research work.

Thirdly, all my sincere thanks and appreciation go to all the members from our Ontario Research Centre for Computer Algebra (ORCCA) lab, Computer Science Department for their invaluable teaching support as well as all kinds of other assistance, and all the members of my thesis examination committee.

Finally, I would like to thank all of my friends around me for their consistent encouragement and would like to show my heartiest gratefulness to my family members for their continued support.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

System of non-linear equations and inequations have much practical importance in many fields such as theoretical physics, chemistry, and robotics. Solving this type of systems means describing the common solutions of the polynomial equations and inequations defining the corresponding system. Since the number of solutions of such systems grows exponentially with the number of unknowns, this process is hard in both contexts of numerical methods and computer algebra methods. The latter scenario, to which this work subscribes, is even more challenging due to the infamous phenomenon of expression swell.

The implementation of non-linear polynomial system solvers is a very active research area. It has been stimulated during the past ten years by two main progresses. Firstly, methods for solving such systems have been improved by the use of so-called modular techniques and asymptotically fast polynomial arithmetic. Secondly, the democratization of supercomputing, thanks to hardware acceleration technologies (multicores, general purpose graphics processing units) creates the opportunity to tackle harder problems.

One central issue in the implementation of polynomial system solvers is the elimination of redundant expressions that frequently occur during intermediate computations, with both numerical methods and computer algebra methods. Factor refinement, also known as coprime factorization, is a popular technique for removing repeated factors among several polynomials, while square-free factorization is used to remove repeated factors within a given polynomial. Coprime factorization has many applications in number theory and polynomial algebra, see the papers [8, 9] and the web site `http://cr.yp.to/coprimes.html` by Bernstein.

Algorithms for coprime factorization have generally been designed with algebraic complexity as the complexity measure to optimize. However, with the evolution of

computer architecture, parallelism and data locality are becoming major measures of performance, in addition to the traditional ones, namely serial running time and allocated space. This imposes to revisit many fundamental algorithms in computer science, such as those coprime factorization.

In this thesis, we revisit the factor refinement algorithms of Bach, Driscoll and Shallit [6] based on quadratic arithmetic. We show that their augment refinement principle leads to a highly efficient algorithm in terms of parallelism and data locality. Our approach takes advantage of the paper "Parallel Computation of the Minimal Elements of a Poset" [19] by Leiserson, Li, Moreno Maza and Xie. Our theoretical results are confirmed by an experimentation, which is based on a multicore implementation in the `Cilk++` concurrency platform [10, 14, 20, 22, 26]. For problems on integers, we use the GMP library [1] while for problems on polynomials, we use the "Basic Polynomial Algebra Subroutines (BPAS)" library [30].

We also revisit the *GCD-free basis* algorithm of Dahan, Moreno Maza, Schost and Xie [15]. Their algorithm, which is serial, is nearly optimal in terms of algebraic complexity. However, our theoretical study combined with experimental results from the literature [13, 29] suggest that this algorithm is not appropriate for a parallelization targeting multicore architecture.

This thesis is organized as follows. Chapter 2 introduces background materials of this research work. Chapter 3 then presents existing serial algorithms for factor refinement and GCD-free basis with their working principles as well as complexity estimates. It is out of the scope of this thesis to give an exhaustive description of the vast amount of previous work; we only provide details for algorithms that we will use. Chapter 4 describes details of parallel factor refinement and GCD-free basis algorithms with estimates of work, span, parallelism and cache complexity. Some implementation issues of parallel algorithms are presented in Chapter 5. Chapter 6 contains our experimental results, performance analysis and benchmarking of the parallel algorithms. Application of subproduct tree techniques for the parallel computation of GCD-free basis is presented in Chapter 7 with some analysis as well as a discussion on the challenges toward an implementation. Finally, conclusion of this thesis and future work appear in Chapter 8.

Chapter 2

Background

This section gathers background materials which are used throughout this thesis. Section 2.1 is a brief introduction to polynomial system solving, which motivates the work reported in this thesis. Section 2.2 is a review of the notion of *square-free factorization*, which is one of the two "simplification techniques" for polynomial systems that are discussed in this thesis. The other one is *coprime factorization*, to which the rest of this document is dedicated and which is introduced in Chapter 3. Section 2.3 is a review of asymptotically fast algorithms for univariate polynomial evaluation and interpolation. It follows closely the presentation in Chapter 10 of [33]. Sections 2.4 and 2.5 describe our two models of computations: the *fork-join parallelism model* and the *idealized cache model*, which were proposed in the papers [21, 22]. Here, we follow closely, the lecture notes of the UWO course CS9624-4435 available at `http://www.csd.uwo.ca/~moreno/CS9624-4435-1011.html`.

2.1 Systems of polynomial equations and inequations

A *system of polynomial equations and inequations* is defined as

$$\begin{cases} f_1(x) = 0 \\ \quad\vdots \\ f_m(x) = 0 \\ h_1(x) \neq 0 \\ \quad\vdots \\ h_s(x) \neq 0 \end{cases} \tag{2.1}$$

where $x = (x_1, x_2, \ldots, x_n)$ refers to n variables, and $f_1, \ldots, f_m, h_1, \ldots, h_s$ are multivariate polynomials in x with coefficients in a field $\mathbb{K}$. Let L be a field extending $\mathbb{K}$. For instance $\mathbb{K}$ and L could be the fields of real and complex numbers, respectively. A *solution over* L of System (2.1) is a tuple $z = (z_1, z_2, \ldots, z_n)$ with coordinates in L such that we have

$$f_1(z) = f_2(z) = \cdots = f_m(z) = 0 \text{ and } h_1(z) \neq 0 \text{ and} \cdots \text{ and } h_s(z) \neq 0. \quad (2.2)$$

A first difference between the case of linear equations and that of polynomial equations is the fact that a non-linear system may have a number of solutions which is both finite and greater than one. Actually, this is often the case in practice, in particular for *square* systems, that is, for systems with as many equations as variables. When finite, this number of solutions, counted with multiplicities, can be as large as d^n, when d is the common total degree of $f_1, f_2, \ldots, f_m$. This fact leads to another difference: "describing" the solutions of System (2.1) is a process that requires a number of arithmetic operations in L which is (at least) exponential in n, the number of variables.

There are two different ways of computing the solutions of System (2.1). The first one relies on so-called *numerical methods* which compute numerical approximations of the possible real or complex solutions. These methods have the advantage of being quite fast in practice while flexible in accuracy by using iterative methods. However, due to rounding errors, numerical methods are principally uncertain, often unstable in an unpredictable way; sometimes they do not find all solutions and have difficulties with overdetermined systems. Some of these approaches are presented in [16, 17, 28].

The alternative way is to represent the solutions symbolically. More precisely, the solution set is decomposed into so-called *components* such that each component is represented by a polynomial system of a special kind, called a *regular chain* [5, 25, 27], which possesses a triangular shape and remarkable properties. For this reason, such a decomposition is called a *triangular decomposition* of the input system. Methods computing triangular decompositions are principally exact. However, they are practically more costly than numerical methods. As a result, they are applicable only to relatively "small" input systems. Nevertheless, they are applicable to any polynomial system of any dimension and for zero-dimensional systems they can precisely provide the number of complex solutions (counted with multiplicities) and specify which solutions have real coordinates.

The practical efficiency as well as the theoretical time complexity of triangular

decompositions depend on various techniques for removing superfluous expressions (components, polynomial factors, ...) during the computations. For instance, if a, b, c are three polynomials which satisfy the following system of inequations

$$a\,b \neq 0, \quad b\,c \neq 0, \quad c\,a \neq 0,$$

one may want to replace it simply by

$$a \neq 0, \quad b \neq 0, \quad c \neq 0,$$

since they are both equivalent and the second one is simpler. As a result, algorithms which do this type of simplification are very important. There are two well-developed techniques for simplifying systems of equations and inequations: one is square-free factorization [7, 35] and another one is *coprime factorization* [6, 15]. The former one is described in next section and the latter one is in next chapter.

2.2 Square-free factorization

Square-free factorization is a first step in many factorization algorithms. It factors non-square-free polynomials in terms of square-free factors that are relatively prime. It can separate factors of different multiplicities, but not factors with the same multiplicity. Formal definitions of square-free factorization are summarized hereafter. For a more complete presentation, please refer to Chapter 14 in [33].

Definition 1. *Let $\mathbb{D}$ be a UFD (unique factorization domain) and $P \in \mathbb{D}[x]$ be a non-constant univariate polynomial. The polynomial P is said square-free if for every non-constant polynomial $Q \in \mathbb{D}[x]$, the polynomial Q^2 does not divide P.*

If $\mathbb{D}$ is a field and $P \in \mathbb{D}[x]$ is square-free, it is easy to show that there exists pairwise different monic irreducible polynomials $P_1, P_2, \ldots, P_k$ such that their product equals $P/\mathrm{lc}(P)$, where $\mathrm{lc}(P)$ denotes the leading coefficient of P. We observe that $P_1, P_2, \ldots, P_k$ are necessarily *pairwise coprime*, that is, for all P_i, P_j, with $1 \leq i < j \leq k$, there exists $A_i, A_j \in \mathbb{D}[x]$ such that $A_i P_i + A_j P_j = 1$ holds. Since $\mathbb{D}$ is a field, using the (extended) Euclidean Algorithm, this latter property is equivalent to the fact that for all P_i, P_j, with $1 \leq i < j \leq k$, the polynomials P_i, P_j have no common factors other than elements of $\mathbb{D}$.

Definition 2. *Let $\mathbb{K}$ be a field and P be a non-constant univariate polynomial $P \in \mathbb{K}[x]$. A square-free factorization of P consists of pairwise coprime polynomials $P_1, P_2, \ldots, P_k$ such that we have $P = P_1 \cdot P_2^2 \ldots P_k^k$ and each P_i is either 1 or a square-free non-constant polynomial.*

It is easy to see that a square-free factorization of P is obtained from the irreducible factors $F_1, \ldots, F_e$ of P by taking for P_i the product of the F_j's such that F_j^i divides P but F_j^{i+1} does not divide P. We illustrate the above definitions with a few examples.

Example 1. *Let $P = (x+1) \in \mathbb{K}[x]$. From Definition 1, the polynomial P is square-free, because there is no non-constant polynomial $Q \in \mathbb{K}[x]$ such that Q^2 divides P. On the other hand, assume $P = x^2+4x+4 \in \mathbb{K}[x]$. From Definition 1, the polynomial Q is not square-free, because, for $Q = (x+2) \in \mathbb{K}[x]$, the polynomial Q^2 divide P.*

Example 2. *Let* $P = x^{15} + 55x^{14} + 1400x^{13} + 21868x^{12} + 234290x^{11} + 1822678x^{10} + 10629552x^9 + 47283632x^8 + 161614309x^7 + 424015067x^6 + 845928448x^5 + 1258456700x^4 + 1348952000x^3 + 981360000x^2 + 432000000x + 86400000 \in \mathbb{R}[x]$, *where $\mathbb{R}$ is the field of real numbers. Then, a square-free factorization of P is given by*

$$P = (x+1)(x+2)^2(x+3)^3(x+4)^4(x+5)^5,$$

where we have $P_1 = (x+1), P_2 = (x+2), P_3 = (x+3), P_4 = (x+4)$, and $P_5 = (x+5)$.

There are several well-known algorithms for computing square-free factorization of a univariate polynomial over a field (and more generally for multivariate polynomials over a field). For instance, Yun proposed an algorithm for computing square-free factorization of univariate polynomials over a field of characteristic zero in 1976 [35]. Yun's algorithm is a basic building block for other polynomial factorization algorithms and it is described in Algorithm 1. An asymptotic upper bound of the algebraic complexity of this algorithm is stated in the following proposition.

Proposition 1. *The cost (number of bit operations) of Algorithm 1 is $O(k^4(n^2d + nd^2))$, where $d = \text{degree}(P_i)$ (thus assuming that all P_i have the same degree), n is the maximum bit size of a coefficient in $P_1, P_2, \ldots, P_k$ and algorithm used for computing GCDs is a quadratic time algorithm (say the Euclidean Algorithm). On the other hand, if $M(d)$ is a multiplication time (that is, a running time estimate in terms of field operations for multiplying two polynomials in $\mathbb{K}[x]$ of degree less than d) then the cost becomes $O(k^2 M(d) \log(d))$ operations in $\mathbb{K}$, where $M(d) \in O(d \log(d) \log\log(d))$ operations in $\mathbb{K}$ for FFT-based multiplication.*

Algorithm 1: `Square-free Factorization`

Input: A polynomial $P \in \mathbb{K}[x]$, where $\mathbb{K}$ is a field of characteristic zero.

Output: A square-free factorization of P, that is, pairwise coprime square-free polynomials $P_1, P_2, \ldots, P_k$ such that $P = P_1 \cdot P_2^2 \ldots P_k^k$ holds in $\mathbb{K}[x]$.

1: $G \leftarrow \gcd(P, \frac{d}{dx}P)$; /* $G = P_2 P_3^2 \ldots P_k^{k-1}$ */

2: $C_1 \leftarrow P/G$; /* $C_1 = P_1 P_2 \ldots P_k$ */

3: $D_1 \leftarrow (\frac{d}{dx}P)/G - \frac{d}{dx}C_1$;
/* $D_1 = P_1 \frac{d}{dx}(P_2) \ldots P_k + 2P_1 P_2 \frac{d}{dx}(P_3) \ldots P_k + \ldots + (k-1)P_1 P_2 \ldots \frac{d}{dx}(P_k)$ */

4: **for** i = 1, step 1, until $C_i = 1$ **do**

5: $P_i \leftarrow \gcd(C_i, D_i)$; /* $C_i = P_i P_{i+1} \ldots P_k$ */

6: $C_{i+1} \leftarrow C_i/P_i$;
/* $D_i = P_i(\frac{d}{dx}(P_{i+1}) \ldots P_k + 2P_{i+1}\frac{d}{dx}(P_{i+2}) \ldots P_k + \ldots + (k-i)P_{i+1} \ldots \frac{d}{dx}(P_k)$ */

7: $D_{i+1} \leftarrow D_i/P_i - \frac{d}{dx}C_{i+1}$;

8: **return** $(P_1, P_2, \ldots, P_k)$;

2.3 Fast polynomial evaluation and interpolation

This section presents asymptotically fast algorithms for univariate polynomial evaluation and interpolation. These are based on the concept of a *subproduct tree*. For a more complete presentation, please refer to Chapter 10 in [33].

2.3.1 Fast polynomial evaluation

Multipoint evaluation of a polynomial (univariate) primarily means evaluating that polynomial at multiple points and can formally be stated as follows. Given a univariate polynomial $P = \sum_{j=0}^{n-1} p_j x^j \in \mathbb{K}[x]$, with coefficients in the field $\mathbb{K}$, and evaluation points $u_0, \ldots, u_{n-1} \in \mathbb{K}$ compute $P(u_i) = \sum_{j=0}^{n-1} p_j u_i^j$, for $i = 0, \ldots, n-1$.

It is useful to introduce the following objects. Define $m_i = x - u_i \in \mathbb{K}[x]$, and $m = \prod_{0 \le i < n}(x - u_i)$. We consider the evaluation map

$$\chi : \begin{array}{ccc} \mathbb{K}[x]/\langle m \rangle & \longrightarrow & \mathbb{K}^n \\ f & \longmapsto & (f(u_0), \ldots, f(u_{n-1})) \end{array}$$

We observe that $\mathbb{K}[x]/\langle m \rangle$ and $\mathbb{K}^n$ are vector spaces of dimension n over $\mathbb{K}$. Moreover, χ is a $\mathbb{K}$-linear map, which is a bijection as soon as the evaluation points $u_0, \ldots, u_{n-1}$ are pairwise distinct.

Problem 1. *(Multipoint Evaluation) Given $n = 2^k$ for some $k \in \mathbb{N}$, $f \in \mathbb{K}[x]$ of degree less than n, and point set $u_0, \ldots, u_{n-1} \in \mathbb{K}$, compute*

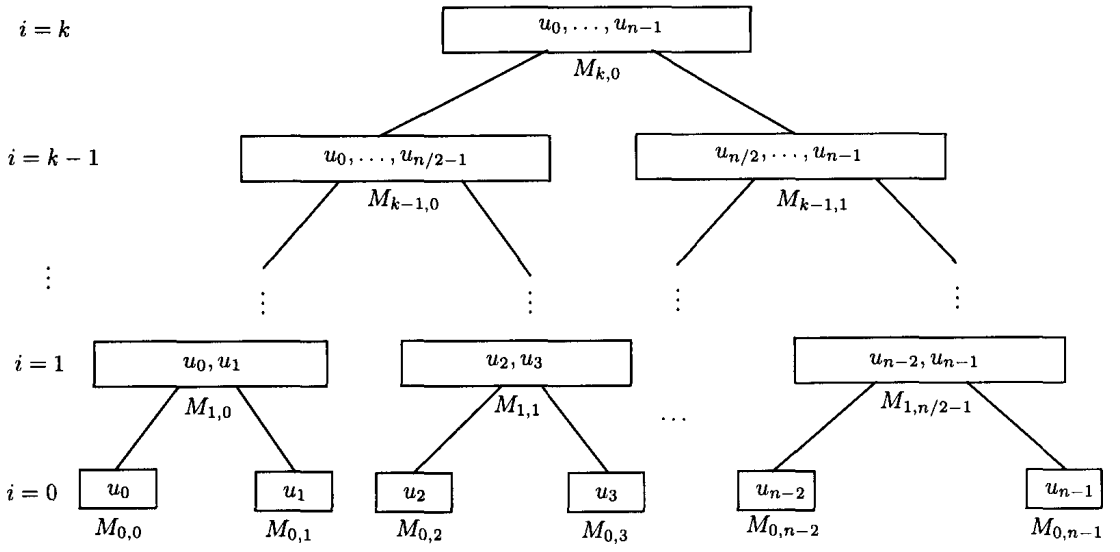

Figure 2.1: Subproduct tree for the multipoint evaluation algorithm.

$$\chi(f) = (f(u_0), \ldots, f(u_{n-1})).$$

This takes $O(n^2)$ operations in $\mathbb{K}$ using *Horner*'s rule n times, but this can be done at a much cheaper cost by means of the subproduct tree method described below.

The idea of the multipoint evaluation using subproduct tree method is to split the point set $u_0, \ldots, u_{n-1}$ into two halves of equal cardinality and to proceed recursively with each of the two halves. This leads to a binary tree of depth $\log_2(n)$ having the points $u_0, \ldots, u_{n-1}$ as leaves. Figure 2.1 illustrates this evaluation scheme.

Algorithm 2: SubproductTree

Input: $m_0 \leftarrow (x - u_0), \ldots, m_{n-1} \leftarrow (x - u_{n-1}) \in \mathbb{K}[x]$ with $u_0, \ldots, u_{r-1} \in \mathbb{K}$ and $n = 2^k$ for $k \in \mathbb{N}$.

Output: $M_{i,j} = \prod_{0 \le l < 2^i} m_{j \cdot 2^i + l}$ for $0 \le i \le k$ and $0 \le j < 2^{k-i}$.

1: **for** $j = 0$ to $n - 1$ **do**
2: $\quad M_{0,j} \leftarrow m_j$;
3: **for** $i = 1$ to k **do**
4: $\quad$ **for** $j = 0$ to $2^{k-i} - 1$ **do**
5: $\quad\quad M_{i,j} \leftarrow M_{i-1,2j} \cdot M_{i-1,2j+1}$;

Let $m_i = x - u_i$ as above, and define

$$M_{i,j} = m_{j\cdot 2^i} \cdot m_{j\cdot 2^i+1} \cdots m_{j\cdot 2^i+(2^i-1)} = \prod_{0\le l<2^i} m_{j\cdot 2^i+l}$$

for $0 \le i \le k = \log_2(n)$ and $0 \le j < 2^{k-i}$. So each $M_{i,j}$ can be represented as a subproduct tree of 2^i products and the recursive definitions of this is as follows.

$$M_{0,j} = m_j \quad \text{and} \quad M_{i+1,j} = M_{i,2j} \cdot M_{i,2j+1}.$$

Here, $M_{i,j}$ means the elements at the j-th node from the left at level i in the subproduct tree shown in Figure 2.1. The algorithm which generates this tree is shown in Algorithm 2.

Proposition 2. *Algorithm 2 takes $O(M(n)\log_2(n))$ operations in $\mathbb{K}$, where M is a multiplication time.*

The computation of these subproducts $(M_{i,j})$ is used as the precomputation stage of the multipoint evaluation algorithm that evaluates the polynomial at points $u_0, \ldots, u_{n-1}$ by traversing the tree in a top-down manner and Algorithm 3 is used to accomplish this.

Algorithm 3: `TopDownTraverse`$(f, M_{k,h})$

Input: Polynomial $f \in \mathbb{K}[x]$ with degree smaller than $n = 2^k$ for some $k \in \mathbb{N}$, pairwise different values $u_0, \ldots, u_{n-1} \in \mathbb{K}$, and $M_{i,j} = \prod_{0\le l<2^i} m_{j\cdot 2^i+l}$ for $0 \le i \le k$ and $0 \le j < 2^{k-i}$ from Algorithm 2.

Output: $f(u_0), \ldots, f(u_{n-1}) \in \mathbb{K}$.

Comment `The algorithm is called with TopDownTraverse`$(f, M_{k,0})$.

1: **if** $n = 1$ **then** `// the degree of` f `is less than` n
2: ⌊ **return** f;
3: $r_0 \leftarrow f \mod M_{k-1,2h}$;
4: $r_1 \leftarrow f \mod M_{k-1,2h+1}$;
5: `TopDownTraverse`$(r_0, M_{k-1,2h})$; `/* Compute` $r_0(u_0), \ldots, r_0(u_{n/2-1})$ `*/`
6: `TopDownTraverse`$(r_1, M_{k-1,2+1})$; `/* Compute` $r_1(u_{n/2}), \ldots, r_1(u_{n-1})$ `*/`
7: **return** $r_0(u_0), \ldots, r_0(u_{n/2-1}), r_1(u_{n/2}), \ldots, r_1(u_{n-1})$;

Proposition 3. *The number of operations required in $\mathbb{K}$ for Algorithm 3 is $O(M(n)\log_2(n))$.*

Finally, Algorithm 4 combines Algorithm 2 and 3 for multipoint evaluation to evaluate the polynomial at points $u_0, \ldots, u_{n-1}$.

Proposition 4. *Algorithm 4 takes $O(M(n)\log_2(n))$ operations in $\mathbb{K}$.*

Algorithm 4: `MultipointEvaluation`

Input: Polynomial $f \in \mathbb{K}[x]$ with degree smaller than $n = 2^k$ for some $k \in \mathbb{N}$, pairwise different values $u_0, \ldots, u_{n-1} \in \mathbb{K}$.

Output: $f(u_0), \ldots, f(u_{n-1}) \in \mathbb{K}$.

1: Call Algorithm 2 with input $(x - u_0), (x - u_1), \ldots, (x - u_{n-1})$ to compute the subproducts $M_{i,j}$;
2: Call Algorithm 3 with input f, the points u_i, and the subproducts $M_{i,j}$;
3: **return** the result of Algorithm 3;

Algorithm 5: `LinearCombination`$(M_{k,v}, n)$

Input: $u_0, \ldots, u_{n-1}, c_0, \ldots, c_{n-1} \in \mathbb{K}$, where $n = 2^k$ for some $k \in \mathbb{N}$, and the polynomial $M_{i,j} = \prod_{0 \le l < 2^i} m_{j \cdot 2^i + l}$ for $0 \le i \le k$ and $0 \le j < 2^{k-i}$ from Algorithm 2.

Output: $\sum_{0 \le i < n} c_i \frac{m}{x - u_i} \in \mathbb{K}[x]$ where $m = (x - u_0) \ldots (x - u_{n-1})$.

Comment The algorithm is called with `LinearCombination`$(M_{k,0}, n)$.

1: **if** $n = 1$ **then** // n is the total number of points
2: ⌊ **return** c_0;
3: $r_0 \leftarrow$ `LinearCombination`$(M_{k-1,2v}, n/2)$; /* Recursive call to compute $r_0 = \sum_{0 \le i < n/2} c_i \frac{M_{k-1,0}}{x - u_i}$ */
4: $r_1 \leftarrow$ `LinearCombination`$(M_{k-1,2v+1}, n/2)$; /* Recursive call to compute $r_1 = \sum_{n/2 \le i < n} c_i \frac{M_{k-1,1}}{x - u_i}$ */
5: **return** $M_{k-1,1} r_0 + M_{k-1,0} r_1$;

2.3.2 Fast polynomial interpolation

In broad terms, polynomial interpolation usually means reconstructing a polynomial from its values at a given set of points. This process can be formally stated as follows.

Let $u_0, \ldots, u_{n-1}$ be pairwise distinct points (i.e. values) in the field $\mathbb{K}$ and let $v_0, \ldots, v_{n-1}$ be another sequence of values in $\mathbb{K}$. Then, there exists a unique polynomial $f \in \mathbb{F}[x]$ of degree less than n such that f takes the values v_i at the point u_i for all i. Moreover, this polynomial is given by

$$f = \sum_{0 \le i < n} v_i s_i \frac{m}{x - u_i}$$

where

$$m = (x - u_0) \cdots (x - u_{n-1}) \quad \text{and} \quad s_i = \prod_{j \neq i} \frac{1}{u_i - u_j}.$$

This s_i can be computed using the derivative of m (denoted by m') by performing

one multipoint evaluation of m' at n points for the given m from the following equation discussed in [34].

$$m'(u_i) = \frac{m}{x-u_i} \mid_{x=u_i} = \frac{1}{s_i}$$

Algorithm 5 and 6 state the whole procedure for computing f from the two sequences $u_0, \ldots, u_{n-1}$ and $v_0, \ldots, v_{n-1}$.

Algorithm 6: `FastInterpolation`

Input: $u_0, \ldots, u_{n-1} \in \mathbb{K}$ such that $u_i - u_j$ is a unit for $i \neq j$, and $v_0, \ldots, v_{n-1} \in \mathbb{K}$, where $n = 2^k$ for some $k \in \mathbb{N}$.

Output: The unique polynomial $f \in \mathbb{K}[x]$ of degree less than n such that $f(u_i) = v_i$ for $0 \leq i < n$.

1: Call Algorithm 2 with input $m_0 = x - u_0, \ldots, m_{n-1} = x - u_{n-1}$ to compute the polynomial $M_{i,j} = \prod_{0 \leq l < 2^i} m_{j \cdot 2^i + l}$ for $0 \leq i \leq k$ and $0 \leq j < 2^{k-i}$;
2: $m \leftarrow M_{k,0}$;
3: Call Algorithm 3 with input $f = m', u_0, \ldots, u_{n-1}$, and $M_{i,j}$ to evaluate m' at $u_0, \ldots, u_{n-1}$;
4: **for** $i = 0$ to $n - 1$ **do**
5: $\quad s_i \leftarrow \frac{1}{m'(u_i)}$;
6: Call Algorithm 5 with input $u_0, \ldots, u_{n-1}, v_0 s_0, \ldots, v_{n-1} s_{n-1}$, and the $M_{i,j}$;
7: **return** the result of Algorithm 5;

Proposition 5. *Algorithm 5 correctly computes the result with $O(M(n)\log_2(n))$ operations in $\mathbb{K}$ and Algorithm 6 (which relies on Algorithm 5) runs correctly within $O(M(n)\log_2(n))$ operations in $\mathbb{K}$, where $M(n)$ is the number of multiplication operations required for the polynomial of degree $n = \sum_{0 \leq i < r} \text{degree}(m_i)$.*

2.4 The fork-join parallelism model

The `Cilk++` concurrency platform [10, 14, 20, 22, 26] provides a simple theoretical model called the *fork-join parallelism model* or *dag (direct acyclic graph) model* of multithreading for parallel computation. This model represents the execution of a multithreaded program as a set of nonblocking threads denoted by the vertices of a dag, where the dag edges indicate dependencies between instructions. See Figure 2.2.

In the `Cilk++` terminology, a thread is a maximal sequence of instructions that ends with a `spawn`, `sync`, or `return` statement. These statements are used to denote respectively:

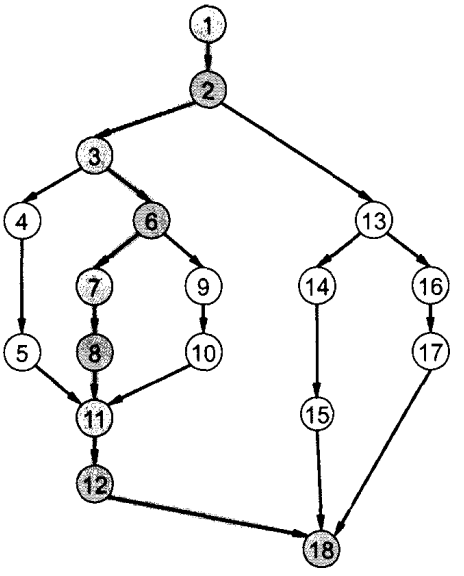

Figure 2.2: A directed acyclic graph (dag) representing the execution of a multithreaded program. Each vertex represents an instruction while each edge represents a dependency between instructions.

- an *execution flow forking*,
- a *synchronization point*, at which currently running threads must join before the execution flow proceeds further,
- the return point of a function.

A correct execution of a `Cilk++` program must meet all the dependencies in the dag, that is, a thread cannot be executed until all the depending treads have completed. The order in which these dependent threads will be executed on the processors is determined by the *scheduler*.

`Cilk++`s scheduler executes any `Cilk++` computation in a nearly optimal time, see [22] for details. From a theoretical viewpoint, there are two natural measures that allow us to define parallelism precisely, as well as to provide important bounds on performance and speedup which are discussed in the following subsections.

2.4.1 The work law

The first important measure is the *work* which is defined as the total amount of time required to execute all the instructions of a given program. For instance, if each instruction requires a unit amount of time to execute, then the work for the example dag shown in Figure 2.2 is 18.

Let T_P be the fastest possible execution time of the application on P processors.

Therefore, we denote the work by T_1 as it corresponds to the execution time on 1 processor. Moreover, we have the following relation

$$T_p \geq T_1/P, \tag{2.3}$$

which is refered as the *work law*. In our simple theoretical model, the justification of this relation is easy: each processor executes at most 1 instruction per unit time and therefore P processors can execute at most P instructions per unit time. Therefore, the *speedup* on P processors is at most P since we have

$$T_1/T_P \leq P. \tag{2.4}$$

2.4.2 The span law

The second important measure is based on the program's *critical-path length* denoted by T_∞. This is actually the execution time of the application on an infinite number of processors or, equivalently, the time needed to execute threads along the longest path of dependency. As a result, we have the following relation, called the *span law*:

$$T_P \geq T_\infty. \tag{2.5}$$

2.4.3 Parallelism

In the fork-join parallelism model, *parallelism* is defined as the ratio of work to span, or T_1/T_∞. Thus, it can be considered as the average amount of work along each point of the critical path. Specifically, the speedup for any number of processors cannot be greater than T_1/T_∞. Indeed, Equations 2.4 and 2.5 imply that speedup satisfies

$$T_1/T_P \leq T_1/T_\infty \leq P.$$

As an example, the parallelism of the dag shown i n Figure 2.2 is $18/9 = 2$. This means that there is little chance for improving the parallelism on more than 2 processors, since additional processors will often starve for work and remain idle.

2.4.4 Performance bounds

For an application running on a parallel machine with P processors with work T_1 and span T_∞, the `Cilk++` *work-stealing scheduler* achieves an expected running time as

follows:

$$T_P = T_1/P + O(T_\infty), \tag{2.6}$$

under the following three hypotheses:

- each strand executes in unit time,
- for almost all parallel steps there are at least p strands to run,
- each processor is either working or stealing.

See [22] for details.

If the parallelism T_1/T_∞ is so large that it sufficiently exceeds P, that is $T_1/T_\infty \gg P$, or equivalently $T_1/P \gg T_\infty$, then from Equation (2.6) we have $T_P \approx T_1/P$. From this, we easily observe that the work-stealing scheduler achieves a nearly perfect linear speedup of $T_1/T_P \approx P$.

2.4.5 Work, span and parallelism of classical algorithms

The work, span and parallelism of some of the classical algorithms in the fork-join parallelism model is shown in Table 2.1.

Algorithm	Work	Span	Parallelism
Merge sort	$\Theta(n \log_2(n))$	$\Theta(\log_2(n)^3)$	$\Theta(\frac{n}{\log_2(n)^2})$
Matrix multiplication	$\Theta(n^3)$	$\Theta(\log_2(n))$	$\Theta(\frac{n^3}{\log_2(n)})$
Strassen	$\Theta(n^{\log_2(7)})$	$\Theta(\log_2(n)^2)$	$\Theta(\frac{n^{\log_2(7)}}{\log_2(n)^2})$
LU-decomposition	$\Theta(n^3)$	$\Theta(n \log_2(n))$	$\Theta(\frac{n^2}{\log_2(n)})$
Tableau construction	$\Theta(n^2)$	$\Omega(n^{\log_2(3)})$	$\Theta(n^{0.415})$
FFT	$\Theta(n \log_2(n))$	$\Theta(\log_2(n)^2)$	$\Theta(\frac{n}{\log_2(n)})$

Table 2.1: Work, span and parallelism of classical algorithms.

2.5 Cache complexity

The *cache complexity* of an algorithm aims at measuring the (negative) impact of memory traffic between the cache and the main memory of a processor executing that algorithm. Cache complexity is based on the *ideal-cache model* shown in Figure 2.3. This idea was first introduced by Matteo Frigo, Charles E. Leiserson, Harald Prokop,

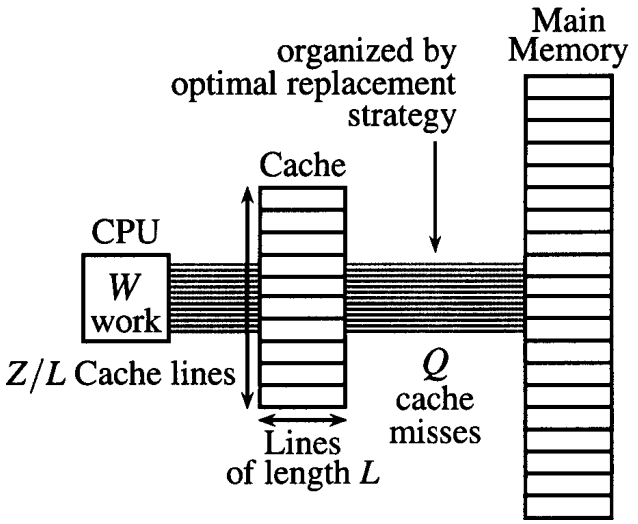

Figure 2.3: The ideal-cache model.

and Sridhar Ramachandran in 1999 [21]. In this model, there is a computer with a two-level memory hierarchy consisting of an ideal (data) cache of Z words and an arbitrarily large main memory. The cache is partitioned into Z/L *cache lines* where L is the length of each cache line representing the amount of consecutive words that are always moved in a group between the cache and the main memory. In order to achieve *spatial locality*, cache designers usually use $L > 1$ which eventually mitigates the overhead of moving the cache line from the main memory to the cache. As a result, it is generally assumed that the cache is *tall* and practically that we have

$$Z = \Omega(L^2).$$

In the sequel of this thesis, the above relation is referred as the *tall cache assumption.*

In the ideal-cache model, the processor can only refer to words that reside in the cache. If the referenced line of a word is found in cache, then that word is delivered to the processor for further processing. This situation is literally called a *cache hit.* Otherwise, a *cache miss* occurs and the line is first fetched into anywhere in the cache before transferring it to the processor; this mapping from memory to cache is called *full associativity.* If the cache is full, a cache line must be evicted. The ideal cache uses the optimal off-line cache replacement policy to perfectly exploit *temporal locality.* In this policy, the cache line whose next access is furthest in the future is replaced [4].

Cache complexity analyzes algorithms in terms of two types of measurements.

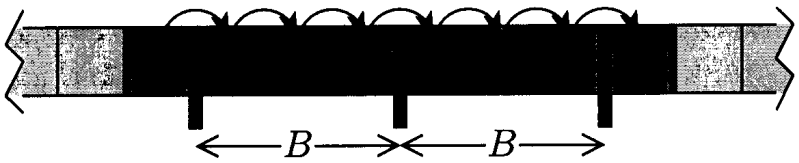

Figure 2.4: Scanning an array of $n = N$ elements, with $L = B$ words per cache line.

The first one is the *work complexity*, $W(n)$, where n is the input data size of the algorithm. This complexity estimate is actually the conventional running time in a RAM model [32]. The second measurement is its *cache complexity*, $Q(n; Z, L)$, representing the number of cache misses the algorithm incurs as a function of:

- the input data size n,
- the cache size Z, and
- the cache line length L of the ideal cache.

When Z and L are clear from the context, the cache complexity can be denoted simply by $Q(n)$.

An algorithm whose cache parameters can be tuned, either at compile-time or at runtime, to optimize its cache complexity, is called *cache aware*; while other algorithms whose performance does not depend on cache parameters are called *cache oblivious*. The performance of cache-aware algorithm is often satisfactory. However, there are many approaches which can be applied to design optimal cache oblivious algorithms to run on any machine without fine tuning their parameters.

Although cache oblivious algorithms do not depend on cache parameters, their analysis naturally depends on the alignment of data block in memory. For instance, due to a specific type of alignment issue based on the size of block and data elements (See Proposition 6 and its proof), the cache-oblivious bound is an additive 1 away from the external-memory bound [24]. However, such type of error is reasonable as our main goal is to match bounds within multiplicative constant factors.

Proposition 6. *Scanning n elements stored in a contiguous segment of memory with cache line size L costs at most $\lceil n/L \rceil + 1$ cache misses.*

PROOF ▷ The main ingredient of the proof is based on the alignment of data elements in memory. We make the following observations.

- Let (q, r) be the quotient and remainder in the integer division of n by L. Let u (resp. w) be the total number of words in a fully (not fully) used cache line. Thus, we have $n = u + w$.

- If $w = 0$ then $(q, r) = (\lfloor n/L \rfloor, 0)$ and the scanning costs exactly q; thus the conclusion is clear since $\lceil n/L \rceil = \lfloor n/L \rfloor$ in this case.
- If $0 < w < L$ then $(q, r) = (\lfloor n/L \rfloor, w)$ and the scanning costs exactly $q + 2$; the conclusion is clear since $\lceil n/L \rceil = \lfloor n/L \rfloor + 1$ in this case.
- If $L \leq w < 2L$ then $(q, r) = (\lfloor n/L \rfloor, w - L)$ and the scanning costs exactly $q + 1$; the conclusion is clear again.

$\triangleleft$

2.6 Multicore architecture

A multicore architecture consists of a multicore processor, which is a single computing component with two or more independent processors called "cores". These cores are the basic units that perform read and execute program instructions. These instructions are ordinary CPU instructions like *add*, *move data*, and *branch*. But, importantly, the multiple cores can execute multiple instructions at the same time, which enhance the overall speed of the program execution in the way of parallel computing. A manycore processor is also a multicore processor in which the number of cores is large enough that traditional multiprocessor techniques are no longer efficient. Manufacturers typically integrate the cores onto a single integrated circuit die, known as a chip multiprocessor or CMP, or onto multiple dies in a single chip package.

The cores in a multicore architecture can be connected *tightly* or *loosely*. For instance, cores may or may not share caches, and they may implement inter-core communication techniques such as message passing or shared memory. Common network topologies are used to interconnect cores, including bus, ring, two-dimensional mesh and crossbar. *Homogeneous* multicore systems include only identical cores, whereas, *heterogeneous* multicore systems have cores which are not identical in practice. Cores on multicore systems may implement architecture features such as instruction level parallelism (ILP), vector processing, SIMD or multithreading, similar to those of single-processor systems.

The advantages of multicore architecture include the fact that *cache coherency* circuitry operates at a much higher clock-rate than in distributed systems where the signals have to travel off-chip. That is, signals between different CPUs (cores) travel shorter distances, and therefore those signals degrade less. As a result, these higher-quality signals with high frequency allow more data to be transferred within a short time period. Moreover, a multicore processor usually uses less power than multiple coupled single-core processors, this is because of the reduced power required to drive

off-chip signals. Furthermore, the cores share some circuitry, like the L2 cache and the interface to the front side bus (FSB). Also, multicore design produces a product with lower risk of design error than devising a new wider core-design.

Although there are lots of advantages of multicores, writing multithreaded programs for this architecture remains quite challenging. Maximizing the utilization of the computing resources in this architecture requires adjustments both to the operating system (OS) support and to existing application software. Also, the performance of multicore processors to execute applications depends on the use of multiple threads within applications. Finally, raw processing power is not the only constraint on system performance. Several processing cores sharing the same system bus and as a result memory bandwidth limits the real-world performance advantage. If a single core is about to consume whole memory-bandwidth, then for the dual-core, it improves only 30% to 70% of its performance. If memory bandwidth is not a problem, upto 90% improvement is possible. Moreover, if communication between the CPUs is the negligible factor, then it would be possible for an application to execute faster on two CPUs than on one dual-core, which would count as much as 100% improvement.

2.7 Systolic arrays

Systolic arrays are matrix-like regular rows of basic data processing units called cells. Each of these cells relies on arriving data from different directions in the array at regular intervals and being combined [12]. The data streams, which are entering and leaving the ports of the array, are generated by *auto-sequencing memory units* called ASMs. In embedded systems, it is also possible that these data streams be input from and/or output to external components.

Matrix multiplication might be a good example of the design of systolic algorithm, where one matrix is fed in a row at a time from the top of the array and is passed down the array. The other matrix is fed in a column at a time from the left hand side of the array and passes from left to right. In order to be seen each processor as a whole row and a whole column, dummy values are often passed in when they are not like so. Finally, the multiplication result is stored in the array and can now be output a row or a column at a time, flowing down or across the array.

Lots of applications of systolic arrays include faster input processing, scalability, high throughput etc. The cells are organized in such a way that it can simultaneously process the input, that is, its processing is faster than the conventional computing architecture. Also, this architecture can easily be extended to many more processors

according to the requirements of the application. Moreover, systolic arrays offer a way to take certain exponential algorithms and use hardware to make them linear.

The disadvantages of systolic arrays include its complicated design and implementation of hardware and software, highly cost of hardware compared to uniprocessor system, highly specialized for particular applications, difficult to build the system etc.

In the perspective of this thesis, systolic arrays are important since they provide the best known work-efficient parallel algorithm for computing GCDs of univariate polynomials [11]. By work-effcient, we mean that the work is the same complexity class as the Euclidean Algorithm. This systolic algorithm for GCDs will be assumed in the results of Chapter 7.

2.8 Graphics processing units (GPUs)

A graphics processing unit (GPU) is a specially designed hardware that accelerates the building of images in a frame buffer targeted for output to a display. This is done in such a special way that it can rapidly manipulate and alter memory in the system. Some of the characteristics of GPUs are as follows:

- GPUs can perform large amounts of floating point computation efficiently since they have lower control overhead.
- They use dedicated functional units for specialized tasks to increase computational speeds.
- GPUs suffer less for memory bandwidth limitations and therefore aims for maximum bandwidth usage.
- Uses some strategy like multiple threads to cope with latency, scheduling of DRAM cycles to minimize idle data-bus time, etc.
- Threads in multithreaded programs are managed by hardware on GPUs rather than software.
- Effective cache design provides higher cache rate.

The applications of GPUs include embedded systems, mobile phones, personal computers, workstations, game consoles, etc. However, in order to achieve the maximum efficiency for these applications, the GPUs are programmed in a very different way than the conventional programming on CPU. Although programming a GPU is not so hard, but it needs detailed knowledge of the architecture of the GPU to exploit the full advantages of the computational power of the GPU. Moreover, it is very necessary to know the data passing techniques through the GPU to devise an efficient algorithm targeted for GPU.

2.9 Random access machine (RAM) model

The RAM is a simple model of computation which is used to measure the run time of an algorithm by counting up the number of steps it takes on a given problem instance. Unlike *Turing machine*, which could not access the memory immediately without accessing all intermediate cells, it can access the arbitrary memory in a single step process. The memory considered, in this model, is unbounded and has the capability to store arbitrarily large integers in each of its memory cells. This model can be programmed in some specified but arbitrary programming language. Some of the properties of this model are as follows:

- Each "simple operation" like addition, subtraction, multiplication, assign, branching, calling, etc. takes exactly 1 time step.
- Loops and procedures are considered to be the composition of many single-step operations.
- Each memory access takes exactly one time step, and we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk, which simplifies the analysis.

A common problem of this model is that it is too simple, that is, these assumptions make the conclusions and analysis too hard to believe in practice. For instance, multiplying two numbers does not have the same cost as adding two numbers, which clearly violates the first assumption of the model. Memory access times also differ greatly depending on whether data are available in cache or on memory or on the disk, which violates the third assumption. However, in spite of having such restrictions, this model does not provide misleading results for the real world problems, since this only assumes a simple abstract model of computation. Furthermore, robustness of the RAM model enables us to analyze algorithms in a machine-independent way.

Chapter 3

Serial Algorithms for Factor Refinement and GCD-free Basis Computation

This chapter brings the background materials which are at the core of this thesis. In Section 3.1, we review the notion of coprime factorization and related concepts, such as factor refinement and GCD-free basis. Serial algorithms computing coprime factorizations are recalled in Sections 3.2 and 3.3. The parallelization of those algorithms will be discussed in the remaining chapters.

3.1 Factor refinement and GCD-free basis

Suppose we have obtained a partial (that is, non-necessarily irreducible) factorization of a univariate polynomial $P(x)$ of degree n with coefficients in a finite field $\mathbb{K}$, say $P(x) = P_1(x)P_2(x)\ldots P_j(x)$. A *refinement* of this factorization is a more "complete" factorization

$$P(x) = \prod_{1 \leq i \leq k} Q_i(x)^{e_i}$$

with $e_i \geq 1$ and where the $Q_i(x) \neq 1$ are pairwise relatively prime, $k \geq 2$ and each P_i, for $1 \leq i \leq j$, writes as a product of some of the Q_i's, for $1 \leq i \leq k$.

In order to unify the presentation for both polynomials and integers, the formal definition below considers a Unique Factorization Domain (UFD) as underlying ring.

Definition 3. *Let $\mathbb{D}$ be a unique factorization domain (UFD, for short). Let m_1,*

$m_2, \ldots, m_r$ *be elements of* $\mathbb{D}$ *and let* m *be their product. Let also* $n_1, n_2, \ldots, n_s$ *be elements of* $\mathbb{D}$. *We say* $n_1, n_2, \ldots, n_s$ *a GCD-free basis whenever* $\gcd(n_i, n_j) = 1$ *for all* $1 \leq i < j \leq s$. *Let* $e_1, e_2, \ldots, e_s$ *be positive integers. We say that the pairs* $(n_1, e_1), (n_2, e_2), \ldots, (n_s, e_s)$ *form a* refinement *of* $m_1, m_2, \ldots, m_r$ *if the following three conditions hold:*

(*i*) $n_1, n_2, \ldots, n_s$ *is a GCD-free basis,*

(*ii*) *for every* $1 \leq i \leq r$ *there exists non-negative integers* $f_1, \ldots, f_s$ *such that we have* $\prod_{1 \leq j \leq s} n_j^{f_j} = m_i$,

(*iii*) $\prod_{1 \leq i \leq s} n_i^{e_i} = m$ *holds.*

When this holds, we shall also say that $n_1, n_2, \ldots, n_s$ *is a GCD-free basis of* $m_1, m_2, \ldots, m_r$. *Finally, whenever* (*i*) *and* (*iii*) *hold, we also say that* $(n_1, e_1), (n_2, e_2), \ldots, (n_s, e_s)$ *is a coprime factorization of* m *and we will often write* ${n_1}^{e_1}, {n_2}^{e_2}, \ldots, {n_s}^{e_s}$ *instead of* $(n_1, e_1), (n_2, e_2), \ldots, (n_s, e_s)$.

Let us make this definition clear using an example with the ring $\mathbb{Z}$ of the integer numbers.

Example 3. *Suppose* $m_1 = 2, m_2 = 6, m_3 = 7, m_4 = 10, m_5 = 15, m_6 = 21, m_7 = 22, m_8 = 26$, *and then* $m = 151351200$. *Suppose also* $n_1 = 11, n_2 = 13, n_3 = 3, n_4 = 7, n_5 = 5, n_6 = 2$ *and* $e_1 = 1, e_2 = 1, e_3 = 3, e_4 = 2, e_5 = 2, e_6 = 5$. *Then from Definition 3 we get that*

(*i*) $(11, 1), (13, 1), (3, 3), (7, 2), (5, 2), (2, 5)$ *is a refinement of* 2, 6, 7, 10, 15, 21, 22, 26,

(*ii*) $11, 13, 3, 7, 5, 2$ *is a GCD-free basis of* $2, 6, 7, 10, 15, 21, 22, 26$,

(*ii*) $(11, 1), (13, 1), (3, 3), (7, 2), (5, 2), (2, 5)$ *is a coprime factorization of* 151351200.

3.2 Quadratic algorithms for factor refinement

The history of factor refinement is long. Research on this topic goes back (at least) to 1974 with Collins who proposed a factor refinement algorithm for polynomials [18]. This result is also noted by von zur Gathen in 1986 [33] for univariate polynomials. An algorithm for multivariate polynomials was proposed by Paul Wang in 1980 [31].

For the case of the integers, a factor refinement algorithm is proposed in [6], based on the following natural elementary procedure. Given a partial factorization of an integer m, say $m = m_1 m_2$, we compute $d = \gcd(m_1, m_2)$ and write

$$m = (m_1/d)(d^2)(m_2/d).$$

Then this process is continued until all the factors are relatively prime. This method is also used for the general case of more than two inputs, say $m = m_1 m_2 \dots m_k$.

Following [6], let us denote by $size(n)$ the number of bits in the binary representation of n. Then, we have:

$$size(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + \lfloor \log_2 |n| \rfloor & \text{if } n > 0. \end{cases}$$

Algorithm 7: `Refine`

Input: Positive integers $m_1, m_2, \dots, m_r \geq 2$; let m be the product of $m_1, m_2, \dots, m_r$.

Output: List of pairs of positive integers $L = [(n_1, e_1), (n_2, e_2), \dots, (n_s, e_s)]$ such that $(n_1, e_1), (n_2, e_2), \dots, (n_s, e_s)$ is a refinement of $m_1, m_2, \dots, m_r$.

comment The algorithm maintains a list L of pairs of positive integers (n_i, e_i) such that $m = \prod_{1 \leq i \leq k} n_i^{e_i}$, where $k \geq 2$ and $e_i \geq 1$.

1: **initialize** $n_i \leftarrow m_i, e_i \leftarrow 1$, a list $L \leftarrow [(n_i, e_i) \mid 1 \leq i \leq r]$.
2: **while** there remains $i < j$ with $\gcd(n_i, n_j) \neq 1$ **do**
3: $\quad d \leftarrow \gcd(n_i, n_j)$;
4: $\quad$ remove pairs $(n_i, e_i), (n_j, e_j)$ from L;
5: $\quad$ add the pairs $(n_i/d, e_i), (d, e_i + e_j), (n_j/d, e_j)$ to L, except for those pairs containing 1 as their first entry;
6: **output** List $L = [(n_1, e_1), \dots, (n_s, e_s)]$.

The authors used the "plain complexity" model where multiplication of m by n is done in $O(size(m)size(n))$ bit operations, the integer remainder $r = m - qn$, $0 \leq r < n$ is computed within $O(size(m/n)size(n))$ bit operations and the GCD computation of $d = \gcd(m, n)$ is performed within $O(size(m/d)size(n))$ bit operations, when $m \geq n$ holds. It is emphasized in [6] that one theorem provided in the paper shows that the output of the algorithm does not depend on the order in which the pairs are examined.

Proposition 7. *Algorithm 7 uses $O(size(m)^3)$ bit operations.*

Algorithm 8: `PairRefine`

Input: Positive integers $m_1, m_2 \geq 2$.

Output: List $L = [(n_i, e_i) \mid n_i \neq 1]$ such that $(n_1, e_1), (n_2, e_2), \ldots, (n_s, e_s)$ is a refinement of m_1, m_2.

1: **initialize** $n_1 \leftarrow m_1, n_2 \leftarrow m_2, e_1 \leftarrow e_2 \leftarrow 1$, a list $L \leftarrow [(n_1, e_1), (n_2, e_2)]$.
2: **while** there remains i with both $n_i, n_{i+1} \neq 1$ **do**
3: $\quad d \leftarrow \gcd(n_i, n_{i+1})$;
4: $\quad$ replace the pairs (n_i, e_i) and (n_{i+1}, e_{i+1}) with $(n_i/d, e_i)$ and $(n_{i+1}/d, e_{i+1})$;
5: $\quad$ insert the pair $(d, e_i + e_{i+1})$ as the new $(i+1)$-th pair;
6: **output** List of pairs $L = [(n_i, e_i) \mid n_i \neq 1]$.

Algorithm 9: `AugmentRefinement`

Input: A positive integer $m_{j+1} \geq 2$ and a list of pairs of positive integers $L_j = [(n_1, e_1), \ldots, (n_s, e_s)]$ such that $n_1, n_2, \ldots, n_s$ is a GCD-free basis, where $n_i \geq 2$ and $e_i \geq 1$.

Output: A refinement of $m_{j+1}, n_1^{e_1}, n_2^{e_2}, \ldots, n_s^{e_s}$.

1: **initialize** $(m, e) \leftarrow (m_{j+1}, 1), L_{j+1} \leftarrow$ empty list.
2: **while** L_j not empty and $m \neq 1$ **do**
3: $\quad (n, f) \leftarrow \texttt{First}(L_j)$;
4: $\quad$ **if** $n \neq 1$ **then**
5: $\quad\quad L' \leftarrow \texttt{PairRefine}((m, e), (n, f))$;
6: $\quad\quad L_{j+1} \leftarrow \texttt{Concat}(L_{j+1}, \texttt{Rest}(L'))$;
7: $\quad\quad (m, e) \leftarrow \texttt{First}(L')$;
8: $\quad L_j \leftarrow \texttt{Rest}(L_j)$;
9: $L_{j+1} \leftarrow \texttt{Concat}(L_{j+1}, \texttt{Rest}(L_j), (m, e))$;
10: **output** List of pairs $(n_i, e_i) \in L_{j+1}$ with $n_i \geq 2$ and $e_i \geq 1$.

In addition, analyzing the data locality of this algorithm (Algorithm 7) leads to the following observation. At lines 4 and 5, the two pairs (n_i, e_i) and (n_j, e_j) are replace by $(n_i/d, e_i)$, $(d, e_i + e_j)$, $(n_j/d, e_j)$ which will be re-scanned in a later iteration of the while-loop. In the worst case, the list pairs L of is scanned $O(n)$ times leading to $O(n^2/L)$ cache misses, using an ideal cache with L words per cache line.

The running time complexity can be improved by keeping tracks of the pairs in an ordered list such that only pairs that are adjacent in the list can have a nontrivial GCD. This modified algorithm is presented in [6]. It relies on two procedures:

- Algorithm 8 which takes two positive integers $m_1, m_2 \geq 2$ as input and produces a refinement of m_1, m_2 as output, and
- Algorithm 9 which "inserts" a new pair in a coprime factorization.

More precisely, Algorithm 9 takes a coprime factorization $(n_1, e_1), \ldots, (n_s, e_s)$ and a positive integer $m_{j+1} \geq 2$; it returns a refinement of $m_{j+1}, n_1^{e_1}, n_2^{e_2}, \ldots, n_s^{e_s}$.

Proposition 8. *Algorithm 8 uses $O(size(l)^2)$ bit operations if $size(l) = size(m_1) + size(m_2)$.*

Proposition 9. *Algorithm 9 runs within $O(size(m)^2)$ bit operations if m is the product of $m_{j+1}, n_1^{e_1}, n_2^{e_2}, \ldots, n_s^{e_s}$.*

On the other hand, similarly to Algorithm 7, we observe that Algorithm 8 proceeds by repeating a "scan-and-replace" pattern, which does not favor data locality. This procedure is invoked by Algorithm 9 and it is easy to see that the whole process may incur $O(n^2/L)$ cache misses. Algorithm 22 proposed in Chapter 4 will improve this situation. Indeed, while preserving the same algebraic complexity, this latter algorithm incurs only $O(n^2/ZL)$ cache misses, for an idealized cache of size Z, with cache lines of size L.

We conclude this section by stating running time estimates of Algorithm 8 and 9 applied to univariate polynomials over a field, instead of integer numbers, which is also presented in [6].

Proposition 10. *Let m_1 and m_2 be* monic *polynomials in $\mathbb{K}[x]$, of degree d_1 and d_2, with $d_1, d_2 > 0$. Let $d = d_1 + d_2$. Then Algorithm 8 uses $O(d^2)$ operations in $\mathbb{K}$.*

Theorem 1. *Let $m_1, m_2, \ldots m_r$ be* monic *polynomials in $\mathbb{K}[x]$, each of them with positive degree. A* refinement *of $m_1, m_2, \ldots, m_r$ may be calculated by repeated augmentation using $O((\sum \deg m_i)^2)$ operations in $\mathbb{K}$.*

3.3 Fast algorithms for GCD-free basis

There are practical cases where coprime factorization routines are applied to an input sequence of square-free polynomials (or integers) $a_1, a_2, \ldots, a_s$. Under this square-freeness assumption, a quasi-linear time coprime factorization algorithm of univariate polynomial over a field (and more generally over a directed product of fields given by a zero-dimensional regular chain) is proposed in [15] and discussed in this section in the field case.

Let $M(d)$ be a *multiplication time*, that is, an upper bound for the number of field operations performed to multiply two univariate polynomials of degree less than d. See Chapter 8 in [33] for a formal presentation of this notion. Define $\mathrm{logp}(x) = 2\log_2(\mathrm{maximum}\{2, x\})$. These notations are applicable throughout this thesis.

The first algorithm (Algorithm 10) computes the sequence of all $\gcd(p, a_i)$ for a polynomial p and a sequence of polynomials $A = a_1, a_2, \ldots, a_e$ as input.

Algorithm 10: `multiGcd`(f, A)

Input: A polynomial $f \in \mathbb{K}[x]$ and a sequence of square-free polynomials $A = a_1, a_2, \ldots, a_e$ in $\mathbb{K}[x]$.

Output: The sequence of all $\gcd(f, a_i)$ for $1 \leq i \leq e$.

1: $d \leftarrow \sum_{i=1}^{e} \text{degree}(a_i)$;
2: **if** $(\text{degree}(f) \geq d)$ **then**
3: $\quad f \leftarrow f \mod (a_1 a_2 \ldots a_e)$;
4: $(q_1, q_2, \ldots, q_e) \leftarrow (f \mod a_1, f \mod a_2, \ldots, f \mod a_e)$;
5: **return** $\gcd(q_1, a_1), \gcd(q_2, a_2), \ldots, \gcd(q_e, a_e)$;

Algorithm 11: `pairsOfGcd`(A, B)

Input: Sequence of polynomials $A = a_1, a_2, \ldots, a_e$ and $B = b_1, b_2, \ldots, b_s$ in $\mathbb{K}[x]$ such that the elements of A (resp. B) are pairwise coprime.

Output: $\gcd(a_1, b_1), \ldots, \gcd(a_1, b_s), \ldots, \gcd(a_e, b_1), \ldots, \gcd(a_e, b_s)$.

1: Build a subproduct tree called $Sub(a_1, a_2, \ldots, a_e)$ with Algorithm 2 where the root is labelled by the product of $a_1 a_2 \ldots a_e$ and let $f = \texttt{RootOf}(Sub)$;
2: Label the root of Sub by $\texttt{multiGcd}(f, B)$;
3: **for** every node $N \in Sub$, going top-down **do**
4: $\quad$ **if** N is not a leaf and has label g **then**
5: $\quad\quad f_1 \leftarrow \texttt{leftChild}(N)$;
6: $\quad\quad f_2 \leftarrow \texttt{rightChild}(N)$;
7: $\quad\quad$ Label f_1 by $\texttt{multiGcd}(f_1, g)$;
8: $\quad\quad$ Label f_2 by $\texttt{multiGcd}(f_2, g)$;
9: Print the leaf labels in a in-fix traversal of the tree;

Proposition 11. *Algorithm 10 amounts to* $O(M(\text{degree}(f)) + M(d)\log(d))$ *operations in* $\mathbb{K}$, *where* d *is the sum of the degrees of the polynomials in* A.

The second algorithm (Algorithm 11) takes two sequences of polynomials $A = a_1, a_2, \ldots, a_e$ and $B = b_1, b_2, \ldots, b_s$ as input, and computes all pairs $\gcd(a_i, b_j)$, for $1 \leq i \leq e$ and $1 \leq j \leq s$. This algorithm assumes that the polynomials in A (resp. B) are pairwise coprime.

Proposition 12. *Algorithm 11 runs within* $O(M(d)\log(d)^2)$ *operations in* $\mathbb{K}$, *where* d *is the sum of the degrees of the polynomials in* A *and* B.

Algorithm 12: `gcdFreeBasisSpecialCase`(A, B)

Input: Sequence of polynomials $A = a_1, a_2, \ldots, a_e$ and $B = b_1, b_2, \ldots, b_s$ where, in each sequence, all polynomials are square-free and pairwise coprime.

Output: A sequence of polynomials forming a GCD-free basis of A, B.

1: $(g_{i,j})_{1 \le i \le e, 1 \le j \le s} \leftarrow$ `pairsOfGcd`(A, B);
2: **for** $j = 1$ to s **do**
3: $\quad L_j \leftarrow$ `removeConstants`$(g_{1,j}, g_{2,j}, \ldots, g_{e,j})$; `// remove constant polynomials`
4: $\quad \beta_j \leftarrow \prod_{l \in L_j} l$;
5: $\quad \gamma_j \leftarrow b_j$ quotient β_j;
6: **for** $i = 1$ to e **do**
7: $\quad L_i \leftarrow$ `removeConstants`$(g_{i,1}, g_{i,2}, \ldots, g_{i,s})$; `// remove constant polynomials`
8: $\quad \alpha_i \leftarrow \prod_{l \in L_i} l$;
9: $\quad \delta_i \leftarrow a_i$ quotient α_i;
10: **return** `removeConstants`$(g_{1,1}, \ldots, g_{i,j}, \ldots, g_{e,s}, \gamma_1, \gamma_2, \ldots, \gamma_s, \delta_1, \delta_2, \ldots, \delta_e)$; `// remove constant polynomials`

The third algorithm (Algorithm 12) takes two sequences of square-free polynomials $A = a_1, a_2, \ldots, a_e$ and $B = b_1, b_2, \ldots, b_s$ as input and computes a GCD-free basis of A and B. This algorithm assumes that the polynomials in A (resp. B) are pairwise coprime. In Algorithm 12, the subroutine *removeConstants* takes a sequence of polynomials $f_1, \ldots, f_m$ of $\mathbb{K}[x]$ and returns the sequence of the non-constant f_i's in the same order as in $f_1, \ldots, f_m$.

Proposition 13. *Algorithm 12 runs within $O(M(d)\log p(d)^2)$ operations in $\mathbb{K}$, where d is the sum of the degrees of the polynomials in A and B..*

Algorithm 13: `gcdFreeBasis`(A)

Input: Sequence of square free polynomials $A = a_1, a_2, \ldots, a_e$.

Output: A GCD-free basis of A.

1: Build a subproduct tree called $Sub'(A)$ like Algorithm 2 where the root is labelled by the sequence of polynomials A;
2: **for** every node $N \in Sub'$, and from bottom-up **do**
3: $\quad$ **if** N is not a leaf **then**
4: $\quad\quad f_1 \leftarrow$ `leftChild`(N);
5: $\quad\quad f_2 \leftarrow$ `rightChild`(N);
6: $\quad\quad$ Label N by `gcdFreeBasisSpecialCase`(f_1, f_2);
7: **return** the label of `RootOf`(Sub');

The fourth algorithm (Algorithm 13) which is also the top-level algorithm, takes a sequence of non-constant square-free polynomials $A = a_1, a_2, \ldots, a_e$ as input and produces a GCD-free basis of A as output.

Proposition 14. *The total number of field operations of Algorithm 13 is* $O(M(d)\mathrm{logp}(d)^3)$, *where* d *is the sum of the degrees of the polynomials in* A.

Chapter 4

Parallel Algorithms for Factor Refinement and GCD-free Basis Computation

Our research work is concerned with the simplification of systems of polynomial equations and inequations. As mentioned before, the goal of these simplifications is to remove repeated patterns such as common factors among inequations. When implemented efficiently, these techniques may greatly improve the performances of polynomial system solvers.

Algorithms for achieving this goal have been presented in the previous section and as they appear in the literature. However, this presentation and the related papers do not consider these algorithms under the angles of data locality and parallelism. Our proposed project is to fill this gap.

To this end, we take advantage of the work reported in [19] in the context of high-performance computing applied to problems on graphs and hypergraphs, such as transversal hypergraph computation. Another important part of our work would be to determine thresholds between plain and fast algorithms and also between serial and parallel execution. It is well-known that plain and serial base cases are essential to achieve best performances. This is why we shall put effort in optimizing both types of algorithms.

In this chapter, we focus on the problem of computing coprime factorization with a view to improve algorithms recalled in Section 3.2 and based on plain (or quadratic) arithmetic. Our aim is twofold. First, we wish to obtain an efficient algorithm in terms of data locality and parallelism. Secondly, we wish to measure the impact of the *augment refinement* principle in this context. Indeed, remember from Chapter 3

that we have looked at two factor refinement algorithms based on plain arithmetic. The first one is Algorithm 7 whose running time is cubic in the sum of the sizes of the input integers (or polynomials). The second one is Algorithm 9, which is based on the augment refinement principle and whose running time is quadratic in the same measure. For clarity in the sequel of this section, we shall refer to Algorithm 7 as the *naive refinement algorithm.*

We shall see in Section 4.2 that Algorithm 9 leads to an efficient algorithmic solution in terms of data locality and parallelism. Section 4.1 shows that we are much less successful with a similar work based on Algorithm 7. The experimental results of Chapter 6 will confirm the complexity analysis of the present chapter.

One should stress the fact that, if the above conclusion seems a natural outcome, one should be careful with hasty conclusions regarding the parallelization of algorithms that are asymptotically fast in terms of algebraic complexity. Indeed, as we shall argue in Chapter 7, GCD-free basis computation algorithms that are based on asymptotically fast arithmetic cannot lead to successful implementation on multicore architectures for the input sizes that are of practical interest today. Therefore, it was not obvious a priori that Algorithm 9 could lead to a better parallel solution than Algorithm 7. This is why we have chosen to report on both parallelization efforts in this chapter.

4.1 Parallelization based on the naive refinement principle

Our proposed parallelization of Algorithm 7 is presented as Algorithm 17 which uses Algorithms 14, 15, and 16 as its subroutines. Algorithm 17 follows a standard divide and conquer approach: until a base case is reached, the input data set is divided into two parts to which the algorithm is applied recursively, producing coprime factorizations which are then merged. As we shall see, the poor efficiency of Algorithm 17 comes from its merging step which fails to use the augment refinement principle.

In a sake of clarity, we have chosen to describe Algorithm 17 when specialized to computing coprime factorizations of (arbitrary large) integer numbers. However, with very little modifications, this algorithm applies to univariate polynomials as well.

Let us describe Algorithm 17 and its subroutines more precisely. Algorithm 14 takes two sequences of square-free positive integers called $A = l_1, l_2, \ldots, l_k$ and $B = m_1, m_2, \ldots, m_r$ as input and computes all possible pairs of GCDs of A and B as

output. These GCDs are then stored in a two-dimensional array called $G = [G_{i,j} \mid 1 \leq i \leq k,\ 1 \leq j \leq r]$.

Algorithm 14: `GcdOfAllPairsInner`(A, B, G)

Input: 1-D arrays of positive integers $A = l_1, l_2, \ldots, l_k$, $B = m_1, m_2, \ldots, m_r$ and a 2-D array $G = [G_{i,j} \mid 1 \leq i \leq k,\ 1 \leq j \leq r]$.

Output: All possible pairs of $\gcd(l_i, m_j)$ for $1 \leq i \leq k, 1 \leq j \leq r$ with $\gcd(l_i, m_j)$ stored in $G_{i,j}$.

comment C is a global variable equal to a base-case threshold, say 16. Assume $C \geq 2$.

```
1:  if k ≤ C and r ≤ C then
2:      for (i, j) ∈ {1, ..., k} × {1, ..., r} do
3:          G_{i,j} ← gcd(A_i, B_j) ;
4:  else if k > C and r ≤ C then
5:      Divide A into two halves A_1 = l_1, ..., l_{k/2}, A_2 = l_{k/2+1}, ..., l_k ;
6:      spawn GcdOfAllPairsInner(A_1, B, G);
7:      spawn GcdOfAllPairsInner(A_2, B, G);
8:  else if k ≤ C and r > C then
9:      Divide B into two halves B_1 = m_1, ..., m_{r/2}, B_2 = m_{r/2+1}, ..., m_r;
10:     spawn GcdOfAllPairsInner(A, B_1, G);
11:     spawn GcdOfAllPairsInner(A, B_2, G);
12: else
13:     Divide A and B arrays into two halves A_1 = l_1, ..., l_{k/2}, A_2 =
        l_{k/2+1}, ..., l_k; B_1 = m_1, ..., m_{r/2}, B_2 = m_{r/2+1}, ..., m_r;
14:     spawn GcdOfAllPairsInner(A_1, B_1, G);
15:     spawn GcdOfAllPairsInner(A_2, B_2, G);
16:     spawn GcdOfAllPairsInner(A_1, B_2, G);
17:     spawn GcdOfAllPairsInner(A_2, B_1, G);
```

Algorithm 15 is a wrapper allocating work space for holding the output of Algorithm 14 before calling this latter algorithm.

Algorithm 15: `GcdOfAllPairs`(A, B)

Input: Array of positive integers $A = l_1, l_2, \ldots, l_k$ and $B = m_1, m_2, \ldots, m_r$.

Output: All possible pairs of $\gcd(l_i, m_j) \mid 1 \leq i \leq k, 1 \leq j \leq r$.

```
1: Allocate space for a 2-D array G = [G_{i,j} | 1 ≤ i ≤ k, 1 ≤ j ≤ r] ;
2: GcdOfAllPairsInner(A, B, G) ; // Call Algorithm 14
3: return array G ;
```

Algorithm 16 is used for merging two coprime factorizations. More precisely, let $A = l_1, l_2, \ldots, l_k$, $B = m_1, m_2, \ldots, m_r$ be two sequences of pairwise coprime square-

free positive integers. Let also $E = e_1, e_2, \ldots, e_k$ and $F = f_1, f_2, \ldots, f_r$ be two sequences of positive integers. We regard A, E (resp. B, F) as a factor refinement $(l_1, e_1), \ldots, (l_k, e_k)$ (resp. $(m_1, f_1), \ldots, (m_r, f_r)$) of some partial factorization. Then, the output is a factor refinement of the concatenation A, E and B, F such that the elements of the corresponding GCD-free bases are square-free. By concatenation of A, E and B, F, we mean the following factorization:

$$(l_1, e_1), \ldots, (l_k, e_k), (m_1, f_1), \ldots, (m_r, f_r). \tag{4.1}$$

Before we analyze its work and span, we make a few remarks on Algorithm 16:

Input: Observe that the assumption that $l_1, l_2, \ldots, l_k$ and $m_1, m_2, \ldots, m_r$ are square-free is essential to guarantee that the algorithm works properly. That is, the algorithm produces a factor refinement. Moreover the elements of this GCD-free basis of this factor refinement are square-free.

Line 3: A GCD-free basis of the elements occurring in A or B is computed by a call to Algorithm 15 followed by the computations at lines 6 and 13. This GCD-free basis consists of the non-one entries appearing in A', B' or G at Line 18.

Line 6: Observe that p_i divides l_i. Indeed, the integer p_i is the product of the $\gcd(l_i, m_j)$'s which all divide l_i. Moreover, these $\gcd(l_i, m_j)$'s are pairwise coprime, because the $m_1, m_2, \ldots, m_r$ are pairwise coprime too.

Line 13: For a similar reason as above, the integer p_j divides m_j

Line 4: No data races occur in this **parallel_for** loop. The fact that G and H are in row-major layout is essential to reduce cache misses here. Moreover, in practice, the grain size of the `cilk_for` should be large enough (probably 64) such that concurrent threads do not compete to acquire data (cache lines).

Line 10: Matrices G and H are transposed before the second **parallel_for** loop in order to improve data locality.

Line 17: Matrices G and H are transposed back to their original layout so as to perform instructions correctly and with a good data locality.

Finally, the top level algorithm, that is, Algorithm 17, receives a sequence of square-free positive integers $A = m_1, m_2, \ldots, m_k$ as input and generates a factor refinement of this sequence. More precisely, it generates two sequences of positive integers $N = n_1, n_2, \ldots, n_s$ and $E = e_1, e_2, \ldots, e_s$ such that $(n_1, e_1), (n_2, e_2), \ldots, (n_s, e_s)$ is a refinement of $m_1, m_2, \cdots, m_k$ where $n_1, n_2, \ldots, n_s$ are pairwise coprime and square-free as well.

Algorithm 16: `MergeRefinement`(A, E, B, F)

Input: Arrays of positive integers $A = l_1, l_2, \ldots, l_k$, $E = e_1, e_2, \ldots, e_k$, $B = m_1, m_2, \ldots, m_r$ and $F = f_1, f_2, \ldots, f_r$, where A, E (resp. B, F) is regarded as a factor refinement $(l_1, e_1), \ldots, (l_k, e_k)$ (resp. $(m_1, f_1), \ldots, (m_r, f_r)$). We assume that the $l_1, l_2, \ldots, l_k$ (resp. $m_1, m_2, \ldots, m_r$) are square-free and pairwise coprime.

Output: A factor refinement of the concatenation of A, E and B, F. If $C = c_1, c_2, \ldots, c_s$ is the GCD-free basis of this output factor refinement, then $c_1, c_2, \ldots, c_s$ are all square-free.

1: **Allocate** G a $(k \times r)$-array, H a $(k \times r)$-array, A' an k-array and B' an r-array. Both G and H are stored in row-major layout;
2: **parallel_for** $(i, j) \in \{1, \ldots, k\} \times \{1, \ldots, r\}$ **do**
    $H_{i,j} \leftarrow 0$;
3: $G \leftarrow$ `GcdOfAllPairs`(A, B) ; `/* Call Algorithm 15 to construct the GCD table regardless of the exponents. */`
4: **parallel_for** $i = 1$ to k **do** `// For each row`
5: $p_i \leftarrow \prod_{1 \le j \le r} G_{i,j}$;
6: $A'_i \leftarrow l_i / p_i$;
7: **for** $j = 1$ to r **do**
8: **if** $G_{i,j} \neq 1$ **then**
9: $H_{i,j} \leftarrow H_{i,j} + e_i$;
10: Transpose G and H to improve data locality ;
11: **parallel_for** $j = 1$ to r **do** `// For each column`
12: $p_j \leftarrow \prod_{1 \le i \le k} G_{j,i}$;
13: $B'_j \leftarrow m_j / p_j$;
14: **for** $i = 1$ to k **do**
15: **if** $G_{j,i} \neq 1$ **then**
16: $H_{j,i} \leftarrow H_{j,i} + f_j$;
17: Transpose G and H back ;
18: Let s_1, s_2, s_3 be the number of the entries in A', B', G that differ from 1;
19: Allocate two integer arrays C and D, both of size $s_1 + s_2 + s_3$;
20: Write the non-one entries of A', B', G to C in the order they appear;
21: Write the corresponding exponents from E, F, H to D;
22: **return** C, D ;

Algorithm 17: `ParallelFactorRefinement(`A`)`

Input: Array of square-free positive integers $A = m_1, m_2, \ldots, m_k$.

Output: Two arrays of positive integers $N = n_1, n_2, \ldots, n_s, E = e_1, e_2, \ldots, e_s$ such that $(n_1, e_1), (n_2, e_2), \ldots, (n_s, e_s)$ is a refinement of $m_1, m_2, \cdots, m_k$. Thus $n_1, n_2, \ldots, n_s$ are pairwise coprime. Moreover, $n_1, n_2, \ldots, n_s$ are square-free.

```
1: if k = 1 then
2:     return [A], [1] ;
3: else
4:     Divide array A into two parts called A_1 and A_2;
5:     f_1 ← spawn ParallelFactorRefinement(A_1) ;
6:     f_2 ← spawn ParallelFactorRefinement(A_2) ;
7:     sync;
8:     return MergeRefinement(f_1, f_2);
```

Proposition 15 analyzes the parallelism of Algorithm 17 for the fork-join parallelism model under one simplification hypothesis, which is stated below.

Hypothesis 1. *Assume that each arithmetic operation (integer division and integer GCD computation) has a unit cost.*

Remark 1. *The primary goal of Hypothesis 1 is to give a first complexity result on the parallelism of Algorithm 17. For problems of practical interest, the input integers are likely to be large, thus multi-precision integer arithmetic is required and Hypothesis 1 does not hold. However, if M is the maximum size of an input integer, then each subsequent arithmetic operation performed by Algorithm 17 and its subroutines runs in $O(M^2)$ bit operations. Therefore, Hypothesis 1 can still be seen as a first approximation of what really happens. As we shall see with Proposition 17, relaxing Hypothesis 1 still leads to a quadratic work (up to log factors) for Algorithm 16. However, the work is now quadratic in the sum of the binary sizes of the input integers rather than quadratic in the number of input integers.*

Remark 2. *In order to analyze Algorithm 17 and its subroutines, one needs to choose a measure of the input data. Hypothesis 1 suggests the following choices. For Algorithm 17 itself, when applied to an array of square-free positive integers $A = m_1, m_2, \ldots, m_n$, this is simply n, the length of A. For each of Algorithm 15 and Algorithm 16, with its notations, this is $k + r$.*

Proposition 15. *Under Hypothesis 1, for an array of square-free positive integers*

$A = m_1, m_2, \ldots, m_n$ of size n, the work, span, and parallelism of Algorithm 17 are respectively $O(n^2)$, $O(n)$ and $O(n)$.

PROOF ▷ Let us denote by $W_{17}(n)$, $W_{16}(n)$, $W_{15}(n)$ (resp. $S_{17}(n)$, $S_{16}(n)$, $S_{15}(n)$) the work (resp. span) of Algorithms 17, 16 and 15 on input data of order n.

We start our analysis with Algorithm 17. Recall that it proceeds in a divide-and-conquer manner, dividing the input data set into two parts, performing two recursive calls and then merging their results with Algorithm 16. Thus we have:

$$W_{17}(n) \leq 2W_{17}(n/2) + W_{16}(n) \text{ and } S_{17}(n) \leq S_{17}(n/2) + S_{16}(n). \tag{4.2}$$

Algorithm 16 calls Algorithm 15 and performs two consecutive parallel for loops, each of them with a critical path of $O(n)$ (due to the inner for loops).

Thus we have:

$$W_{16}(n) \leq W_{15}(n) + O(n^2) \text{ and } S_{16}(n) \leq S_{15}(n) + O(n). \tag{4.3}$$

Algorithm 15 is just a wrapper allocating work space and calling Algorithm 14. This latter algorithm proceeds in a divide-and-conquer manner. For simplicity we assume that $C = 2$ holds. This does not affect the correctness of the algorithm. Moreover, keeping C arbitrary does not bring much insight on the algorithm since for cases of practical interest n is much larger than C. For simplicity we also assume that n is a power of 2, since we focus on the asymptotic behavior of the work and the span. With these simplification assumptions, the work and span of Algorithm 15 satisfy the following equations:

$$W_{15}(n) \leq 4W_{15}(n/2) + O(1) \text{ and } S_{15}(n) \leq S_{15}(n/2) + O(1). \tag{4.4}$$

From Equation 4.4, we have:

$$W_{15}(n) = O(n^2) \text{ and } S_{15}(n) = O(\log_2(n)).$$

From Equation 4.3, we obtain

$$W_{16}(n) = O(n^2) \text{ and } S_{16}(n) = O(n).$$

Finally, from Equation 4.2, we deduce

$$W_{17}(n) = O(n^2) \text{ and } S_{17}(n) = O(n).$$

This completes the proof of the proposition. $\lhd$

We turn now our attention to cache complexity, considering the **serial elisions** of Algorithm 17 and its subroutines. This means that, from now on,

- we replace by the empty string both keywords **spawn** and **sync** in these algorithms, and
- we replace the keyword **parallel_for** by **for**.

In other words, we analyze the cache complexity of the serial versions of our algorithms.

The results of the paper [23] by Matteo Frigo and Volker Strumpen justifies this approach that reduces the cache complexity analysis of multithreaded algorithms to the cache complexity analysis of their serial counterparts. Successful examples for algorithms like matrix multiplication, 1-D stencil, Gaussian elimination and back substitution are presented in [23]. We leave for future

- to show that this reduction is also valid for Algorithm 22 and its subroutines,
- to adjust the complexity result of Proposition 16 with the multithreaded correction terms yielded by a multithreaded execution.

One should observe that, for the examples of [23], the multithreaded correction terms are always of lower order when compared to the terms coming from the serial analysis. For instance, a multithreaded cache oblivious matrix multiplication incurs $O(n^3/\sqrt{Z} + (Pn)^{1/3}n^2)$ cache misses when executed by the Cilk scheduler on a machine with P processors, each with a cache of size Z. In this case, the serial term is $n^3/\sqrt{Z}$ and the multithreaded correction term is $(Pn)^{1/3}n^2$.

Analyzing the cache complexity of serial (or multithreaded) algorithms requires specifying how data is layed out in memory. This leads to the following hypothesis.

Hypothesis 2. *Assume that there exists a positive integer w such that each integer coefficient of each input array A or B in Algorithms 17, 16 and 15 is stored in w consecutive machine words. We also assume that each input array A or B is* packed, *that is, its successive coefficients occupy consecutive memory slots.*

Remark 3. *The first part of Hypothesis 2 can be seen as a natural consequence of Hypothesis 1. The second part is a standard code optimization technique that can always be enforced whether Hypothesis 1 holds or not. Of course, this brings extra work and implementing this strategy efficiently is crucial. We shall return to this issue in Chapter 5. One can also refer to [19] for an implementation discussion on data packing.*

Our cache complexity analysis focuses on Algorithm 14. Here's the reason. Memory accesses are performed in Algorithms 16 and 14 only, so only these two algorithms are relevant to cache complexity analysis. Moreover, as we shall see, Algorithm 14 is an *efficiency bottleneck* for Algorithm 16, in terms of cache complexity. Based on this, there is no point in putting effort in analyzing Algorithm 16: the result will be at least as bad as the one of Algorithm 14.

Proposition 16. *Under Hypotheses 1 and 2, consider an ideal cache of Z words, with L words per cache-line. Then, for C small enough, for any input of size n, the number of cache misses of Algorithm 14 is $Q(n) \in O(n^2/L + n^2/Z + n^2/Z^2)$. Using the* tall cache assumption, *this becomes $Q(n) \in O(n^2/L)$.*

PROOF ▷ To keep the calculations simple, we assume that w divides L exactly and that the input arrays A and B are aligned. This assumption has no incidence on the complexity class of $Q(n)$.

We use the idealized cache model described in Section 2.5. The key observation is that, when n is small enough, the whole Algorithm 14 can be executed without cache misses other than cold misses, that is, without cache misses other than those necessary to bring once (and only once) the cache lines storing the input and output data into cache. This implies that there exists a positive constant α such that $Q(n)$ satisfies the following relation:

$$Q(n) \leq \begin{cases} n^2/L + n & \text{for } n < \alpha Z \\ 4Q(n/2) + \Theta(1) & \text{otherwise,} \end{cases}$$

provided $C < \alpha Z$. Let us justify the above recurrence relation. The case $n \geq \alpha Z$ is correct simply because of the recursive structure of the algorithm. The case where $n < \alpha Z$ holds (for $n < C$) corresponds to the situation where only cold cache misses occur. In this case, bringing the cache lines with the input arrays A and B is done within $\Theta(n/L + 1)$ cache misses. However, bringing the cache lines for storing the computed n^2 values $G_{i,j}$ amounts to $\Theta(n^2/L + n)$ cache misses. The above recurrence

leads to the following inequality for all $n \geq 2$:

$$\begin{aligned} Q(n) &\leq 4Q(n/2) + \Theta(1) \\ &\leq 4[4Q(n/4) + \Theta(1)] + \Theta(1) \\ &\vdots \\ &\leq 4^k Q(n/2^k) + \sum_{j=0}^{k-1} 4^j \Theta(1) \end{aligned}$$

where $k = \lceil \log_2(n/\alpha Z) \rceil$. Since $n/2^k \leq \alpha Z$, we deduce:

$$\begin{aligned} Q(n) &\leq (n/\alpha Z)^2[(\alpha Z)^2/L + \alpha Z] + \Theta((n/\alpha Z)^2) \\ &= \Theta(n^2/L + n^2/Z + n^2/Z^2). \end{aligned}$$

This completes the proof. $\lhd$

Remark 4. *Proposition 16 is a negative result for the following reason. First of all, we should observe that this asymptotic upper bound can be reached. Indeed, it is easy to construct an example for that. Secondly, in practice, the value of L is in the order of 8 to 16 while that of Z is in the order of 2^{15} to 2^{17}, which justifies the tall cache assumption. Therefore, the ratio between the work and the number of cache misses is L, that is, quite small. Now recall that the penalty for one cache miss in a multicore processor is typically in the order of 100 CPU cycles while a machine word operation may cost less than one CPU cycle, thanks to instruction level parallelism (ILP). Therefore a multithreaded program implementing Algorithm 14 on a multicore architecture may spend much more time in waiting for data transfer than in actual computations. This is clearly not satisfactory.*

We return now the analysis of the work of Algorithm 16. We shall no longer assume that each division or GCD computation has a unique cost. We shall now take into account the size of the operands for estimating the cost of a division or GCD computation. For simplicity, we do this analysis when Algorithm 16 is applied to univariate polynomials over a field, instead of integers, as stated below.

Hypothesis 3. *We assume that the input arrays A and B of Algorithm 16 contains k and r univariate polynomials over a field $\mathbb{K}$. We denote by $|p|$ the degree of a non-zero univariate polynomial p over $\mathbb{K}$. We assume that, for $p_1, p_2 \in \mathbb{K}[x]$ computing the quotient (in the Euclidean division) of p_1 by p_2 or computing a GCD of p_1, p_2 can be done within $O(|p_1||p_2|)$ operations in $\mathbb{K}$.*

Proposition 17. *Let n denote the sum of the degrees of the polynomials in A and B. Then, under Hypothesis 3, Algorithm 16 works within $O(n^2)$ operations in $\mathbb{K}$, up to log factors.*

PROOF ▷ At Line 3, computing the GCD of l_i and m_j can be done within $O(|l_i||m_j|)$ operations in $\mathbb{K}$, for each $1 \leq i \leq k$ and each $1 \leq j \leq r$. Thus the total cost of Line 3 is

$$O\left(\sum_{i=1}^{i=k} |l_i| \sum_{j=1}^{j=r} |m_j|\right) \tag{4.5}$$

operations in $\mathbb{K}$. Observe that, for a fixed $i \in \{1, \ldots, k\}$, the sum of the degrees of $G_{1,1} \ldots, G_{1,r}$ is at most the degree of l_i. Indeed, the input polynomials in A (resp. B) are square-free and pairwise coprime. Therefore, using subproduct tree techniques, all the products at Line 5 can be computed within

$$O\left(\sum_{i=1}^{j=k} |l_i|^2\right). \tag{4.6}$$

operations in $\mathbb{K}$, **up to log factors**. For a reference, see Chapter 10 in [34]. Next, all divisions at Line 6 can be computed within

$$O\left(\sum_{i=1}^{i=k} |l_i|^2\right). \tag{4.7}$$

bit operations. Similarly, the total cost of Lines 12 and 13 is

$$O\left(\sum_{j=1}^{j=r} |m_j|^2\right). \tag{4.8}$$

Therefore, up to log factors, the total cost of Algorithm 16 is

$$O\left(\sum_{i=1}^{i=k} |l_i| \sum_{j=1}^{j=r} |m_j|\right) + O\left(\sum_{i=1}^{j=k} |l_i|^2\right) + O\left(\sum_{j=1}^{j=r} |m_j|^2\right). \tag{4.9}$$

Letting $u = \sum |l_i|$ and $v = \sum |m_i|$, this can be bounded over by $O(uv + u^2 + v^2)$, up to log factors, and the conclusion follows. ◁

Remark 5. *Proposition 17 shows that relaxing Hypothesis 1 still leads to a quadratic work (up to log factors) for Algorithm 16. However, the work is now quadratic in the sum of the sizes of the input data items (integers or polynomials) rather than quadratic in the number of the data items. Therefore, the inefficiency of Algorithm 16 comes primarily for its high cache complexity and not so much from an excess of work.*

Turning to implementation considerations, the situation becomes worse. Indeed, in practice, subproduct tree techniques, which are behind the complexity result of Propo-

sition 17, have several drawbacks. First, they increase memory consumption by a log factor. Secondly, they provide benefits in terms of algebraic complexity only for very large input sizes. Thirdly, they are hard to parallelize on multicore architectures. We shall return to all these implementation considerations in Chapter 7.

4.2 Parallelization based on the augment refinement principle

Algorithm 22 presented in this section is an efficient algorithmic solution in terms of data locality and parallelism for coprime factorization of integers or univariate polynomials. With respect to Algorithm 17, presented in the previous section, the main gain is in terms of cache complexity. To be more specific, consider an ideal cache of Z words, with L words per cache-line. Using the tall cache assumption, for an input data of size of n, Algorithm 22 incurs $O(n^2/ZL)$ cache misses while Algorithm 17 suffers from $O(n^2/L)$ cache misses. This substantial gain is obtained thanks to a cache friendly and memory efficient procedure (Algorithm 21) for merging two coprime factorizations. Before analyzing Algorithm 22, we describe its specifications and those of its subroutines, namely Algorithms 18, 19, 20, and 21.

Algorithm 18 receives two square-free polynomials $a, b \in \mathbb{K}[x]$ and two positive integers e, f. Its output is a factor refinement of a^e, b^f.

The input of Algorithm 19 is a square-free polynomial $a \in \mathbb{K}[x]$, a positive integer e, a sequence of square-free pairwise coprime univariate polynomials $B = (b_1, b_2, \ldots, b_n)$ of $\mathbb{K}[x]$ and a sequence of positive integers $F = (f_1, f_2, \ldots, f_n)$. The output is a factor refinement of $a^e, {b_1}^{f_1}, \ldots, {b_n}^{f_n}$.

Algorithm 20 takes two sequences of square-free pairwise coprime polynomials $A = (a_1, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_r)$ of $\mathbb{K}[x]$ together with two sequences of positive integers $E = (e_1, e_2, \ldots, e_n)$ and $F = (f_1, f_2, \ldots, f_r)$. The output is a factor refinement of ${a_1}^{e_1}, \ldots, {a_n}^{e_n}, {b_1}^{f_1}, \ldots, {b_r}^{f_r}$.

The input and outputs specification of Algorithm 21 are respectively the same as those of Algorithm 20. The difference is that Algorithm 21 is a parallel execution, which uses Algorithm 20 as serial base case.

Finally, the top level algorithm (Algorithm 22) takes a sequence of square-free univariate polynomials $A = (m_1, m_2, \ldots, m_k) \in \mathbb{K}[x]$ as input and generates as output a sequence of univariate polynomials $N = (n_1, n_2, \ldots, n_s) \in \mathbb{K}[x]$ and a sequence

Algorithm 18: `PolyRefine`(a, e, b, f)

Input: Two square-free univariate polynomials $a, b \in \mathbb{K}[x]$ for a field $\mathbb{K}$ and two positive integers e, f.

Output: (c, u, G, V, d, w) where $c, d \in \mathbb{K}[x]$ and $u, w \in \mathbb{N}$ and G is a sequence $(g_1, \ldots, g_s)$ of polynomials of $\mathbb{K}[x]$ and V is a sequence $(v_1, \ldots, v_s)$ of positive integers such that $(c, u), (g_1, v_1), \ldots, (g_s, v_s), (d, w)$ is a factor refinement of $(a, e), (b, f)$.

```
1:  g ← gcd(a, b);
2:  a' ← a quotient g;
3:  b' ← b quotient g;
4:  if g = 1 then
5:      return (a, e, ∅, ∅, b, f) ; // Here ∅ designates the empty sequence
6:  else if a = b then
7:      return (1, 1, (a), (e + f), 1, 1)
8:  else
9:      (ℓ1, e1, G1, V1, r1, f1) ← PolyRefine(a', e, g, e + f);
10:     (ℓ2, e2, G2, V2, r2, f2) ← PolyRefine(r1, f1, b', f);
11:     if ℓ2 ≠ 1 then
12:         G2 ← G2 + (ℓ2) ; // Here + designates sequence concatenation
13:         V2 ← V2 + (e2);
14:     return (ℓ1, e1, G1 + G2, V1 + V2, r2, f2) ;
```

of positive integers $E = (e_1, e_2, \ldots, e_s)$ such that $((n_1, e_1), (n_2, e_2), \ldots, (n_s, e_s))$ is a refinement of $(m_1, m_2, \cdots, m_k)$. Moreover, $n_1, n_2, \ldots, n_s$ are square-free.

Proposition 18 analyzes the parallelism of Algorithm 22 for the fork-join parallelism model under one simplification hypothesis, which is stated below.

Hypothesis 4. *Assume that each polynomial operation (division or GCD computation) involved in Algorithm 22 or its subroutines has a unit cost.*

Remark 6. *As for Proposition 15, our primary goal with Proposition 18 is to give a first complexity result on the parallelism of Algorithm 22. Obviously, Hypothesis 4 is not realistic. However, if D is the maximum degree of an input polynomial and if each arithmetic operation (addition, multiplication, inversion) in $\mathbb{K}$ has a constant bit cost, then each subsequent polynomial operation performed by Algorithm 22 or its subroutines runs in $O(D^2)$ bit operations. Therefore, Hypothesis 4 can still be seen as a first approximation of what really happens.*

Remark 7. *In order to analyze Algorithm 22 and its subroutines, one needs to choose a measure of the input data. Hypothesis 4 suggests the following choice. For each of*

Algorithm 19: `MergeRefinePolySeq`(a, e, B, F)

Input: A square-free polynomial $a \in \mathbb{K}[x]$, a positive integer e, a sequence of square-free pairwise coprime polynomials $B = (b_1, b_2, \ldots, b_n)$ of $\mathbb{K}[x]$ and a sequence of positive integers $F = (f_1, f_2, \ldots, f_n)$.

Output: (ℓ, m, Q, R, S, T) where $\ell \in \mathbb{K}[x]$, $m \in \mathbb{N}$, $Q = (q_1, \ldots, q_s)$ and $S = (s_1, \ldots, s_p)$ are two sequences of polynomials of $\mathbb{K}[x]$, $R = (r_1, \ldots, r_s)$ and $T = (t_1, \ldots, t_p)$ are two sequences of positive integers such that $(\ell, m), (q_1, r_1), \ldots, (q_s, r_s), (s_1, t_1), \ldots, (s_p, t_p)$ is a factor refinement of $a^e, {b_1}^{f_1}, \ldots, {b_n}^{f_n}$.

```
1:  ℓ_0 ← a;
2:  m_0 ← e;
3:  Q ← ∅;
4:  R ← ∅;
5:  S ← ∅;
6:  T ← ∅;
7:  for i from 1 to n do
8:      (ℓ_i, m_i, G_i, V_i, d_i, w_i) ← PolyRefine(ℓ_{i-1}, m_{i-1}, b_i, f_i) ;
9:      Q ← Q + G_i ;
10:     R ← R + V_i ;
11:     if d_i ≠ 1 then
12:         S ← S + (d_i) ;
13:         T ← T + (w_i) ;
14: return (ℓ_n, m_n, Q, R, S, T);
```

Algorithms 18, 19, 20, 21 or 22, the input size is the number of polynomials in the input.

Proposition 18. *Hypothesis 1, for an input data of size n, the work, span, and parallelism of Algorithm 22 are respectively $O(n^2)$, $O(Cn)$ and $O(n/C)$, where C is the threshold* BASESIZE.

PROOF ▷ Let us denote by $W_{22}(n)$, $W_{21}(n)$, $W_{20}(n)$, $W_{19}(n)$, $W_{18}(n)$ (resp. $S_{22}(n)$, $S_{21}(n)$, $S_{20}(n)$, $S_{19}(n)$, $S_{18}(n)$) the work (resp. span) of Algorithms 22, 21, 20, 19 and 18 on input data of order n.

The top-level routine is Algorithm 22. It proceeds in a divide-and-conquer manner, dividing the input data into two equal parts, performing two recursive calls and then merging their results with Algorithm 21. Thus we have:

$$W_{22}(n) \leq 2W_{22}(n/2) + W_{21}(n) \quad \text{and} \quad S_{22}(n) \leq S_{22}(n/2) + S_{21}(n). \tag{4.10}$$

Algorithm 20: `MergeRefineTwoSeq`(A, E, B, F)

Input: Two sequences of square-free pairwise coprime polynomials $A = (a_1, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_r)$ of $\mathbb{K}[x]$ together with two sequences of positive integers $E = (e_1, e_2, \ldots, e_n)$ and $F = (f_1, f_2, \ldots, f_r)$.

Output: (L, M, Q, R, S, T) where $L = (\ell_1, \ldots, \ell_h)$, $Q = (q_1, \ldots, q_s)$ and $S = (s_1, \ldots, s_p)$ are three sequences of polynomials of $\mathbb{K}[x]$, $M = (m_1, \ldots, m_h)$, $R = (r_1, \ldots, r_s)$ and $T = (t_1, \ldots, t_p)$ are three sequences of positive integers such that $(\ell_1, m_1), \ldots, (\ell_h, m_h), (q_1, r_1), \ldots, (q_s, r_s), (s_1, t_1), \ldots, (s_p, t_p)$ is a factor refinement of $(a_1, e_1), \ldots, (a_n, e_n), (b_1, f_1), \ldots, (b_r, f_r)$.

1: $L \leftarrow \varnothing$;
2: $M \leftarrow \varnothing$;
3: $Q \leftarrow \varnothing$;
4: $R \leftarrow \varnothing$;
5: $S_0 \leftarrow B$;
6: $T_0 \leftarrow F$;
7: **for** i from 1 to n **do**
8: $\quad (\ell_i, m_i, Q_i, R_i, S_i, T_i) \leftarrow$ `MergeRefinePolySeq`$(a_i, e_i, S_{i-1}, T_{i-1})$;
9: $\quad Q \leftarrow Q + Q_i$;
10: $\quad R \leftarrow R + R_i$;
11: $\quad$ **if** $\ell_i \neq 1$ **then**
12: $\quad\quad L \leftarrow L + (\ell_i)$;
13: $\quad\quad M \leftarrow M + (m_i)$;
14: **return** (L, M, Q, R, S_n, T_n);

Algorithm 21 also proceeds in a divide-and-conquer manner, dividing the input data into two parts and performing two recursive calls. Then using the output of these two recursive calls as input of two subsequent recursive calls. To keep this analysis simple, we assume that n is a power of 2. However, and on the contrary of what we did in the proof of Proposition 15, we do not set the threshold BASESIZE to 2. The reason is because the costs (algebraic and cache complexity) play a crucial role in the performance of Algorithm 21. This fact will be made more clear within the proof of our cache complexity result, namely Theorem 2. We define $C :=$ BASESIZE and we have

$$W_{21}(n) \leq \begin{cases} W_{20}(n) & \text{for } n < C \\ 4W_{21}(n/2) + \Theta(1) & \text{otherwise,} \end{cases} \tag{4.11}$$

Algorithm 21: `MergeRefinementDNC`(A, E, B, F)

Input: Two sequences of square-free pairwise coprime polynomials $A = (a_1, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_r) \in \mathbb{K}[x]$ together with two sequences of positive integers $E = (e_1, e_2, \ldots, e_n)$ and $F = (f_1, f_2, \ldots, f_r)$.

Output: (L, M, Q, R, S, T) where $L = (\ell_1, \ldots, \ell_h)$, $Q = (q_1, \ldots, q_s)$ and $S = (s_1, \ldots, s_p)$ are three sequences of polynomials of $\mathbb{K}[x]$, $M = (m_1, \ldots, m_h)$, $R = (r_1, \ldots, r_s)$ and $T = (t_1, \ldots, t_p)$ are three sequences of positive integers such that $(\ell_1, m_1), \ldots, (\ell_h, m_h), (q_1, r_1), \ldots, (q_s, r_s), (s_1, t_1), \ldots, (s_p, t_p)$ is a factor refinement of $(a_1, e_1), \ldots, (a_n, e_n), (b_1, f_1), \ldots, (b_r, f_r)$.

1: **if** $n \leq$ BASESIZE or $r \leq$ BASESIZE **then**
2: ⌊ **return** `MergeRefineTwoSeq`(A, E, B, F);
3: **else**
4: Divide A, E, B, and F into two halves called A_1, A_2, E_1, E_2, B_1, B_2, and F_1, F_2, respectively ;
5: $(L_1, M_1, Q_1, R_1, S_1, T_1) \leftarrow$ **spawn** `MergeRefinementDNC`(A_1, E_1, B_1, F_1) ;
6: $(L_2, M_2, Q_2, R_2, S_2, T_2) \leftarrow$ **spawn** `MergeRefinementDNC`(A_2, E_2, B_2, F_2) ;
7: **sync** ;
8: $(L_3, M_3, Q_3, R_3, S_3, T_3) \leftarrow$ **spawn** `MergeRefinementDNC`(L_1, M_1, S_2, T_2) ;
9: $(L_4, M_4, Q_4, R_4, S_4, T_4) \leftarrow$ **spawn** `MergeRefinementDNC`(L_2, M_2, S_1, T_1) ;
10: **sync** ;
11: **return** $\{L_3 + L_4, M_3 + M_4, Q_1 + Q_2 + Q_3 + Q_4, R_1 + R_2 + R_3 + R_4, S_3 + S_4, T_3 + T_4\}$;

Algorithm 22: `ParallelFactorRefinementDNC`(A)

Input: A sequence of square-free polynomials $A = (m_1, m_2, \ldots, m_k) \in \mathbb{K}[x]$.

Output: A sequence of square-free pairwise coprime polynomials $N = (n_1, n_2, \ldots, n_s) \in \mathbb{K}[x]$, and a sequence of positive integers $E = (e_1, e_2, \ldots, e_s)$ such that $(n_1, e_1), (n_2, e_2), \ldots, (n_s, e_s)$ is a factor refinement of $m_1, m_2, \cdots, m_k$.

1: **if** $k < 2$ **then**
2: ⌊ **return** $(m_1), (1)$;
3: **else**
4: Divide A into two subsequences called A_1 and A_2;
5: $(X_1, Y_1) \leftarrow$ **spawn** `ParallelFactorRefinementDNC`(A_1) ;
6: $(X_2, Y_2) \leftarrow$ **spawn** `ParallelFactorRefinementDNC`(A_2) ;
7: **sync**;
8: **return** `MergeRefinementDNC`(X_1, Y_1, X_2, Y_2);

and

$$S_{21}(n) \leq \begin{cases} S_{20}(n) & \text{for } n < C \\ 2S_{21}(n/2) + \Theta(1) & \text{otherwise.} \end{cases} \tag{4.12}$$

Hypothesis 4 implies that $W_{20}(n)$, $W_{19}(n)$, $W_{18}(n)$ fit within $\Theta(n^2)$, $\Theta(n)$, $\Theta(1)$, respectively. Moreover, since Algorithms 20, 19 and 18 are serial, we also have $S_{20}(n)$, $S_{19}(n)$, $S_{18}(n)$ within $\Theta(n^2)$, $\Theta(n)$, $\Theta(1)$, respectively.

Let $k = \lceil \log_2(n/C) \rceil$. Then we have

$$W_{21}(n) \leq O(4^k C^2) = O(n^2) \text{ and } S_{21}(n) \leq O(2^k C^2) = O(Cn).$$

Therefore, from Relation (4.10), we deduce

$$W_{22}(n) \in O(n^2) \text{ and } S_{22}(n) \in O(Cn).$$

This completes the proof. $\lhd$

We observe that, under a "unit cost" hypothesis for the integer (or polynomial) arithmetic, the work, span and parallelism of Algorithm 17 and Algorithm 22 are essentially the same.

We turn now our attention to cache complexity, which will differentiate these two algorithms substantially. As for the algorithms analyzed in Section 4.1, we analyze the cache complexity of the serial versions of our algorithms. With Hypothesis 5, we start by specifying how data is layed out in memory.

Hypothesis 5. *We assume that each input or output sequence in Algorithms 18, 19, 20 or 21 is* packed, *that is, its successive polynomials occupy consecutive memory slots. We also assume that each element of the field* $\mathbb{K}$ *can be stored in one machine word.*

Next, we specify a few helpful notations for establishing our cache complexity results.

Notation 1. *We denote by* $|p|$ *the degree of a non-zero univariate polynomial* p *over* $\mathbb{K}$. *For a finite polynomial sequence* $P = (p_1, \ldots, p_n)$, *we denote by* $|P|$ *the sum of the degrees of* $p_1, \ldots, p_n$.

The purpose of the following hypothesis is to enforce the following property: any polynomial p occurring in any input sequence or in any output sequence of Algorithms 18, 19, 20 or 21 can be stored within $|p|$ machine words.

Hypothesis 6. *All polynomials in any input or output sequence of Algorithms 18, 19, 20 or 21 is not constant and monic. Each integer in any input or output sequence of positive integers is stored in one machine word.*

Remark 8. *As they are currently stated, Algorithms 18, 19, 20 or 21 do not meet Hypothesis 6. However, it is not difficult to modify them in order to satisfy Hypothesis 6. Consider for instance Algorithm 18. The necessary modifications consist in handling the cases $a' \in \mathbb{K}$ and $b' \in \mathbb{K}$ in order to avoid the unnecessary recursive calls at Lines 9 and 10; moreover, if one of the returned polynomials ℓ_1 or r_2 is constant, one can use a Boolean instead.*

Lemma 1. *Under Hypothesis 6, with the notations in the specifications of Algorithm 18, we have*

$$|c| + |G| \leq |a| \quad \text{and} \quad |G| + |d| \leq |b|. \tag{4.13}$$

PROOF ▷ We proceed by induction on the sum of the degrees of the input polynomials of Algorithm 18, that is, $|a| + |b|$. First, we observe that, at Line 4, if $g = 1$ holds, then the conclusion trivially holds. Indeed, in this case, we have $c = a$, $G = \varnothing$ and $d = b$. Similarly, at Line 6, if $a = b$ holds, the conclusion also holds. These two cases cover, in particular, the base case of the induction, that is, $|a| + |b| = 2$.

Now, we assume that g has a positive degree. Thus, we can apply the induction hypothesis to the two recursive calls at Lines 9 and 10, since we have $|a'|+|g| < |a|+|b|$ and thus $|r_1| + |b'| \leq |g| + |b'| < |a| + |b|$.

Next, we shall estimate $|\ell_1|$, $|G_1+G_2|$ and $|r_2|$ at the return point of Algorithm 18, that is, at Line 14. We designate by G_2^0 the value of G_2 after executing Line 10 and before executing Line 11. Then we have:

$$\begin{aligned}
|\ell_1| + |G_1 + G_2| &\leq |\ell_1| + |G_1| + |G_2| && \text{by definition} \\
&\leq |\ell_1| + |G_1| + |G_2^0| + |\ell_2| && \text{by definition} \\
&\leq |a'| + |g| && \text{by induction} \\
&\leq |a| && \text{since } a = a'g.
\end{aligned}$$

Similarly, we have

$$\begin{aligned}
|G_1 + G_2| + |r_2| &\leq |G_1| + |G_2| + |r_2| && \text{by definition} \\
&\leq |G_1| + |\ell_2| + (|G_2^0| + |r_2|) && \text{by definition} \\
&\leq |G_1| + |r_1| + |b'| && \text{by induction} \\
&\leq |g| + |b'| && \text{by induction} \\
&\leq |b| && \text{since } b = b'g.
\end{aligned}$$

This completes the proof. $\lhd$

Lemma 2. *Under Hypothesis 6, with the notations in the specifications of Algorithm 19, we have*

$$|\ell| + |Q| \leq |a| \quad \text{and} \quad |Q| + |S| \leq |B|. \tag{4.14}$$

PROOF $\rhd$ We use the notations of the pseudo-code of Algorithm 19. We have the following inequalities:

$$\begin{aligned}
|\ell_n| + |Q| &\leq |\ell_n| + |G_n| + |G_{n-1}| + \cdots + |G_1| && \text{by definition} \\
&\leq |\ell_{n-1}| + |G_{n-1}| + \cdots + |G_1| && \text{by Lemma 1} \\
&\vdots && \vdots \\
&\leq |\ell_1| + |G_1| && \text{by Lemma 1} \\
&\leq |l_0| && \text{by Lemma 1} \\
&\leq |a| && \text{by definition.}
\end{aligned}$$

With Lemma 1, we also have:

$$|Q| + |S| \leq \sum_{i=1}^{i=n} (|G_i| + |d_i|) \leq \sum_{i=1}^{i=n} |b_i| \leq |B|.$$

This completes the proof. $\lhd$

Lemma 3. *Under Hypothesis 6, with the notations in the specifications of Algorithm 20, we have*

$$|L| + |Q| \leq |A| \quad \text{and} \quad |Q| + |S_n| \leq |B|. \tag{4.15}$$

PROOF ▷ We use the notations of the pseudo-code of Algorithm 20. We have the following inequalities:

$$\begin{aligned} |L| + |Q| &\leq \sum_{i=1}^{i=n} (|\ell_i| + |Q_i|) && \text{by definition} \\ &\leq \sum_{i=1}^{i=n} |a_i| && \text{by Lemma 2} \\ &\leq |A| && \text{by definition} \end{aligned}$$

We also have:

$$\begin{aligned} |Q| + |S_n| &\leq |S_n| + |Q_n| + |Q_{n-1}| + \cdots + |Q_1| && \text{by definition} \\ &\leq |S_{n-1}| + |Q_{n-1}| + \cdots + |Q_1| && \text{by Lemma 2} \\ &\vdots && \vdots \\ &\leq |S_1| + |Q_1| && \text{by Lemma 2} \\ &\leq |S_0| && \text{by Lemma 2} \\ &\leq |B| && \text{by definition.} \end{aligned}$$

This completes the proof. ◁

Proposition 19. *Under Hypothesis 6, for an input of size n, each of the Algorithms 18, 19 and 20 can be run in space $\Theta(n)$ bits.*

PROOF ▷ Lemmas 1, 2 and 3 imply that for an input of size n, the output size is at most n. Recall that by input size or output size, we mean the total number of polynomial coefficients (except the leading coefficients since they are all set to 1). The output contains also sequences of positive integers, the number of those being at most n. Finally, we observe that each of the Algorithms 18, 19 and 20 does not require extra memory space other than:

- the space for the input and output data,
- a constant number of pointers and index variables.

This completes the proof. ◁

Lemma 4. *Under Hypotheses 5 and 6, for an ideal cache of Z words, with L words per cache-line, there exists a positive constant α such that for an input of size n, satisfying $n < \alpha Z$, the number of cache misses of Algorithm 20 is $O(n/L + 1)$.*

PROOF ▷ Indeed, it follows from Proposition 19, that, there exists a positive constant β such that for an input of size n, the memory requirement for running Algorithm 20 is at most βn bits. And those βn bits, up to a constant number of them, are used for storing a number (independent of n) of sequences of polynomials and positive integers. Now recall from Hypotheses 5 that each sequence of polynomials (resp. positive integers) is stored in an array. Finally, recall that loading to cache (resp. writing back to main memory) an array of length ℓ requires $O(\ell/L+1)$ cache misses. This conclusion follows. ◁

Theorem 2. *Under Hypotheses 5 and 6, consider an ideal cache of Z words, with L words per cache-line. Then, for C small enough, for any input of size n, the number of cache misses of Algorithm 21 is $Q(n) = O(n^2/ZL + n^2/Z^2)$. Using the* tall cache assumption, *this becomes $Q(n) \in O(n^2/ZL)$.*

PROOF ▷ We use the idealized cache model described in Section 2.5. It follows from Lemma 4 and the recursive structure of Algorithm 21 that there exists a positive constant α such that $Q(n)$ satisfies the following relation:

$$Q(n) \leq \begin{cases} O(n/L+1) & \text{for } n < \alpha Z \\ 4Q(n/2) + \Theta(1) & \text{otherwise,} \end{cases}$$

provided $C < \alpha Z$. Strictly speaking the above recurrence assumes that each of the input polynomial sequences A and B can be split into two sequences of approximately the same size. When the total number of polynomials in A and B is large, this is likely. When it is small, the condition $n < \alpha Z$ is likely holds and we are "out of the woods". A more formal treatment can be done using standard (but quite involved) proof techniques, as for the parallelization of *quick-sort* algorithm[1].

The above recurrence leads to the following inequality for all $n \geq 2$:

$$\begin{aligned} Q(n) &\leq 4Q(n/2) + \Theta(1) \\ &\leq 4[4Q(n/4) + \Theta(1)] + \Theta(1) \\ &\vdots \\ &\leq 4^k Q(n/2^k) + \sum_{j=0}^{k-1} 4^j \Theta(1) \end{aligned}$$

[1]For details, see the slides of the lecture *Analysis of Multithreaded Algorithms* available at `http://www.csd.uwo.ca/~moreno/CS9624-4435-1011.html`

where $k = \lceil \log_2(n/\alpha Z) \rceil$. Since $n/2^k \leq \alpha Z$, we deduce:

$$\begin{aligned} Q(n) &\leq (n/\alpha Z)^2(\alpha Z/L + 1) + \Theta((n/\alpha Z)^2) \\ &= O(n^2/ZL + n^2/Z^2). \end{aligned}$$

This completes the proof. $\lhd$

Chapter 5

Implementation Issues

In this chapter, we will discuss some important implementation issues of the algorithms presented in the previous chapter. Effective use of caches of multicore machines to exploit the benefit of data locality is one of the most important issues among them to be considered. The implementation of multi-dimensional array in available programming languages does not guarantee the possible storing of all array elements in consecutive memory locations. It possibly stores every elements of each row in consecutive memory slots but not one row after another. As a result, this implementation is not cache friendly in terms of data locality. In order to achieve the benefit of spatial locality, we have to implement multi-dimensional arrays in such a way that array elements are stored in consecutive memory locations. Alternatively, we can say that we have to pack the data for eventually storing it in consecutive memory locations. The non-packed and packed version of a two-dimensional array are shown in Figure 5.1.

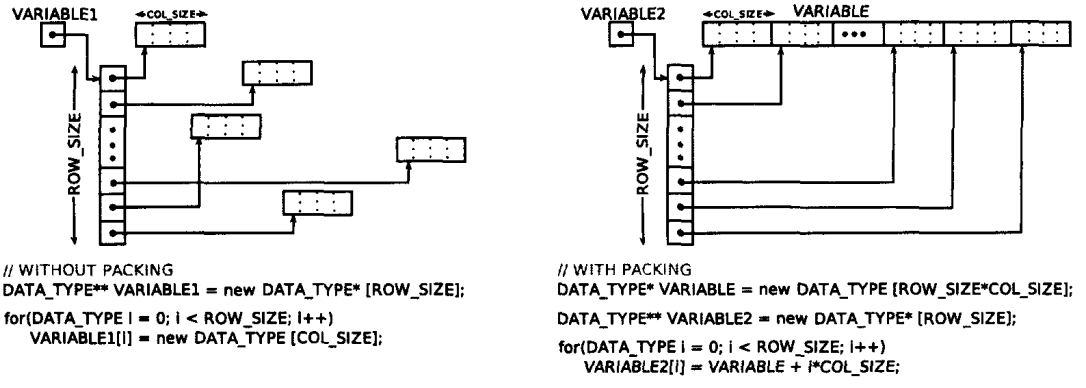

Figure 5.1: Demonstration of unpacking and packing.

Another issue that may impact performances negatively is unbalanced data traffic. This phenomenon can be explained as follows. Suppose that, on a multicore architecture, several threads are making requests to the memory controller. If the requests of one of those threads deal with much larger chunks than the other threads requests, then the memory controller may give a higher priority to this particular thread, resulting in possible data starvation of the others.

Returning to the algorithms of the previous chapter, the low-level routines are GCD calculations and divisions which may take as input data of very different sizes. These routines are called by higher level procedures which generally take as input a set of data items. Proceeding in a divide-and-conquer manner, these procedures divide the data set in "tentatively" equal parts. When a base case is reached, low-level routines operate on the data items. Consequently, unbalanced data divisions may substantially increase the burdened span of the actual application. Indeed, for a given unbalanced data division, threads handling recursive calls getting smaller portion may starve. Therefore, for the call which made the unbalanced data division, the span might become essentially equal to work!

In order to deal with this issue, we ensure balanced data division, by sorting the data set before division. This prepossessing may increase the work but experimentally, this extra work compensated by the reduction of the burdened span. This data traffic balancing for polynomial inputs is illustrated in Figure 5.2. Here in (a), input to the algorithm for computing GCD and division is $10, 9$ in its left sub-problem and $1, 2$ in its right sub-problem. Clearly, data traffic for the left sub-problem in memory is more than that of of the right one, if data was not already in cache. Due to thread starvation, the right sub-problem may not complete before the left sub-problem. Consequently, the core executing the right sub-problem is not fully available to help processing work that could be spawned from the left sub-problem. If, in addition, the left sub-problem itself performs unbalanced data division, performances might reduce further.

On the other hand, in (b), inputs to both the left and right sub-problems are balanced and they should be completed possibly within the same time without without thread starvation.

For certain applications, this data traffic balancing may increase the work. Suppose that the low-level routines (here division and GCD computation) have a running time which is between linear and quadratic in the size of the input. Suppose also that every data item has to be compared every other. Then data traffic balancing may not have a negative impact on the work. However, if this assumption does not hold,

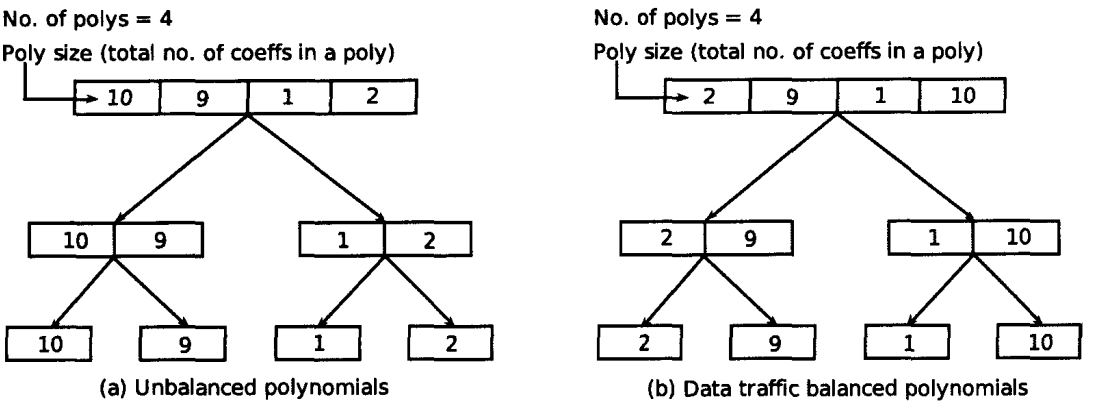

Figure 5.2: Balancing polynomials for data traffic during divide-and-conquer.

for instance when searching the minimal elements of a partially ordered set [19], then data traffic balancing might increase the work. But, as mentioned above, this seems not to be the case in our application.

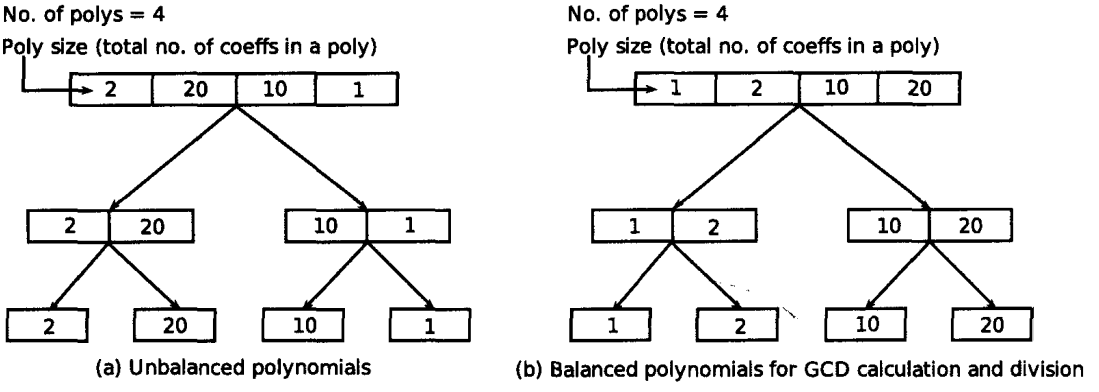

Figure 5.3: Balancing polynomials for GCD calculation and division during divide-and-conquer.

Finally, choosing the base case size for the implementation of multithreaded algorithms is one of the vital issues. Choosing these thresholds are guided by two motivations. First, the need of reducing parallelization overheads, that is, reducing the cost of thread management (creation, synchronization, etc.). In this point of view, large thresholds mean less parallelization overheads, but also less parallelism. Thus a trade-off has to be found. A second motivation is to reduce cache misses in algorithms processing data by block decomposition, as for blocked matrix multiplication. In that point of view, small block sizes mean that the corresponding sub-problems fit

in cache (limiting cache misses to cold misses only) but small block sizes also imply overhead in dividing the data set (due to alignments and related issues). Therefore, another trade-off has to be found here.

The effect of these implementation issues is presented in next chapter titled experimental results.

Chapter 6

Experimental Results

Chapter 4 describes proposed parallel algorithms followed by some important implementation issues discussed in Chapter 5. In this chapter, we will restrict ourselves to verifying the theoretical analysis of these algorithms by presenting some experimental results. All the algorithms are implemented in the `Cilk++` concurrency platform [10, 14, 20, 22, 26] with a software library called "Basic Polynomial Algebra Subroutines (BPAS)" implemented by Dr. Yuzhen Xie and Dr. Marc Moreno Maza in our laboratory (ORCCA), as stated before, and execute them on two machines. The speedup estimates (scalability analysis) of these algorithms are obtained by `Cilkview`, feature of `Cilk++`, after executing them in an Intel(R) Xeon(R) (64 bit) Machine, with CPU (E7340) Speed 2.40GHz, 128.0 GB of RAM, and having a total of 16 Cores available in one of the sharcnet clusters [3]. On the other hand, all timings of the proposed algorithms and `Maple` [2] functions (`Maple`'s built-in function `ifactor` for `int` and non-built-in function `GcdFreeBasis_mod` for polynomial) are obtained by running these algorithms in an Intel(R) Core(TM) i7 (64 bit) Machine, with CPU (870) Speed 2.93GHz, 8.0 GB of RAM with 8 Cores configured in the ORCCA Lab. Moreover, the results of timings are generated with the average value of 5 executions.

The input polynomials supplied in these algorithms for experimentation are of two degree classes: naive refinement based parallel factor refinement algorithm takes polynomials each of degree 60 while augment refinement based parallel factor refinement algorithm takes those of degree 150. However, operations on coefficients, for both cases as well as for `Maple`, are done in modulo a small prime, say 5.

6.1 Integers of type int inputs

Figure 6.1 and 6.2 show the speedup estimates and execution times, respectively, for augment refinement based parallel algorithm described in Section 4.2 in case of `int` type data as input. It is observed that, the speedup is almost linear with the number of cores of the machine. The reason is that, its merging of two refined outputs is efficient in terms of data locality and parallelism which is also presented in Section 4.2. Moreover, its running times are also improved compared to `Maple` even if we consider these for a single processor.

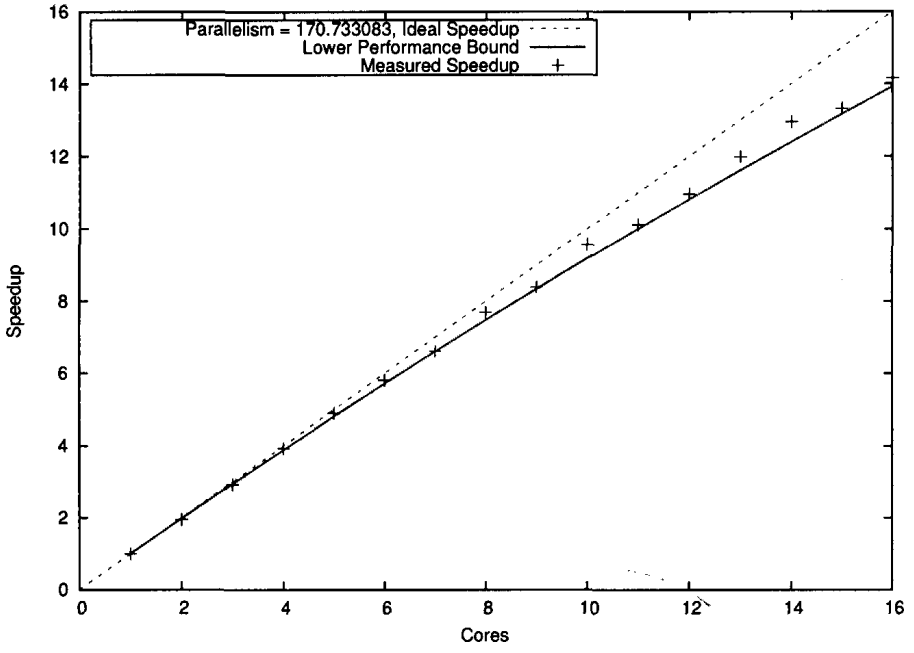

Figure 6.1: Scalability analysis of the augment refinement based parallel factor refinement algorithm for 200,000 `int` type inputs by `Cilkview`.

6.2 Polynomial type inputs

The following two figures (Figure 6.3 and 6.4) show the speedup estimates and execution times, respectively, with dense polynomials as input in case of naive refinement based parallel factor refinement algorithm described in Section 4.1. The speedup shown here is not linear, because the merging of two refined polynomials in this algorithm is not efficient in terms of data locality and it is also verified in that sec-

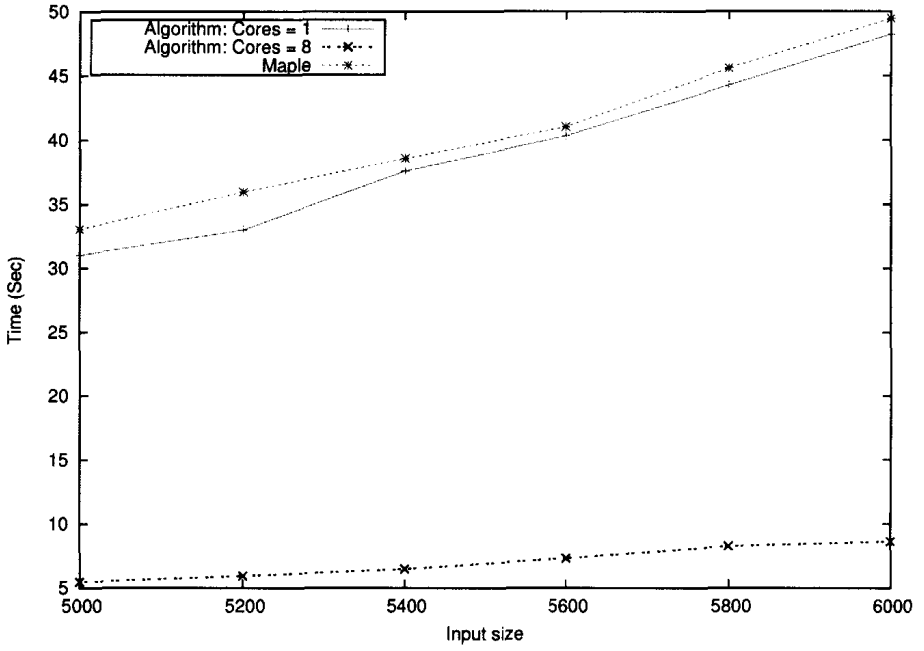

Figure 6.2: Running time comparisons of the augment refinement based parallel factor refinement algorithm for `int` type inputs.

tion. However, the timings compared to `Maple` are improved, because current `Maple` procedure for refinement is not parallel as well as data locality is not considered.

On the other hand, the experimental results of the augment refinement based parallel algorithm described in Section 4.2, with dense polynomials as input, are shown in Figure 6.5 and 6.6. It is also observed that, the speedup is almost linear with the number of cores of the machine similar to the inputs of type `int`. The reason is that, its merging of two refined outputs is efficient in terms of data locality and parallelism like the case of `int` type. Moreover, its running times are also improved compared to `Maple` even if we consider this for a single processor.

Our final goal is to analyze the performance of the augment refinement based parallel algorithm described in Section 4.2 when the provided input pattern is already a GCD-free basis. The analysis from Figure 6.7 and 6.8, generated for this pattern, shows that the speedup is still almost linear and its running time outperforms `Maple`.

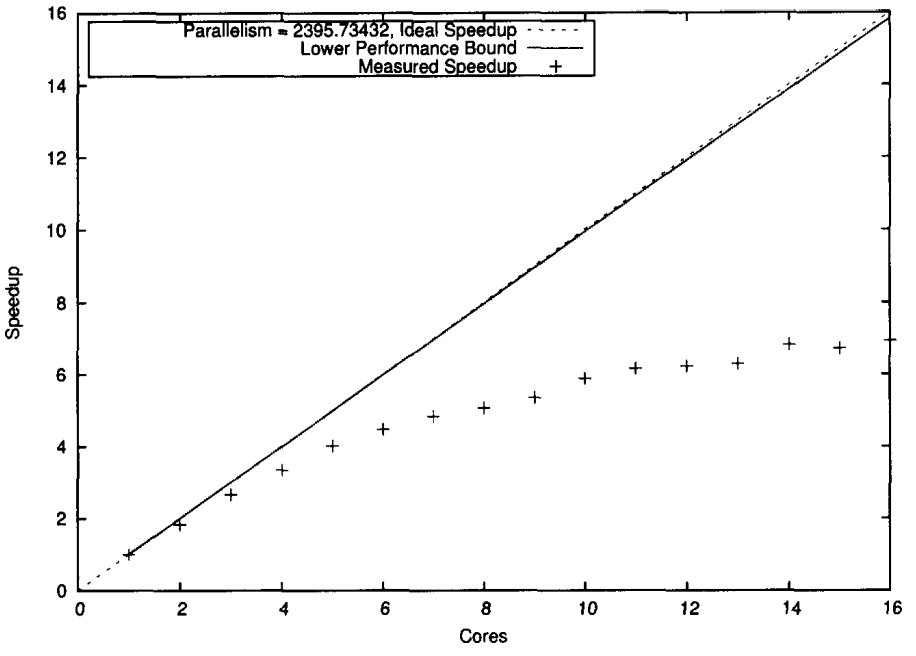

Figure 6.3: Scalability analysis of the naive refinement based parallel factor refinement algorithm for 4,000 dense square-free univariate polynomials by `Cilkview`.

6.3 Integers of type my_big_int inputs (work in progress)

Factor refinement of large integer (GMP) type data does not show linear speedup due to the structure of this data type. It actually uses `structure` of pointers to `int` to hold large integer numbers, divided into chunks, which are not necessarily stored in consecutive memory slots. But, storing the data possibly in consecutive memory slots is very necessary to achieve linear speedup, because, it preserves spatial data locality in caches. Without this, there are lots of cache misses while executing the program and a significant impact on whole program performance. In order to recover this problem of locality in case of large integer numbers, we need to design a user defined large integer data type called `my_big_int`, where all the chunks of integer numbers are possibly stored in consecutive memory slots. We are currently working on designing the data structure of this data type and its implementation prior to applying this to the factor refinement algorithm to examine its performance.

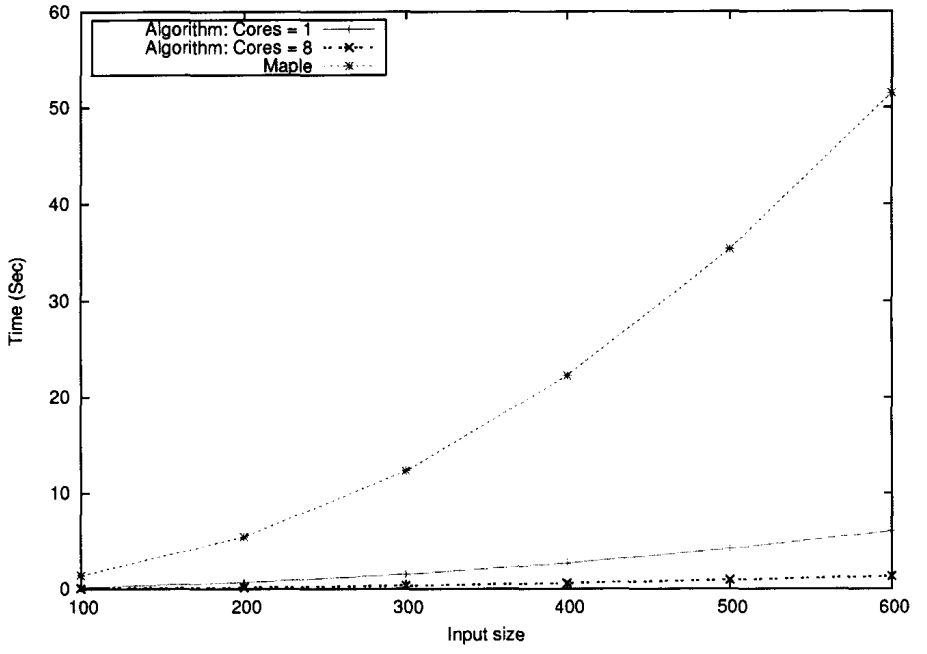

Figure 6.4: Running time comparisons of the naive refinement based parallel factor refinement algorithm for dense square-free univariate polynomials.

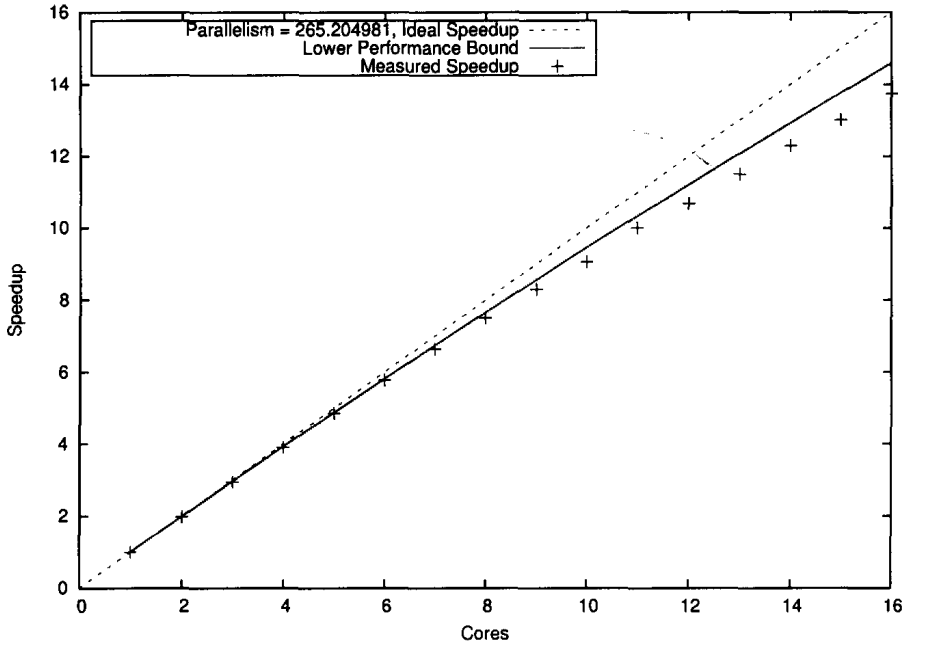

Figure 6.5: Scalability analysis of the augment refinement based parallel factor refinement algorithm for 4,000 dense square-free univariate polynomials by `Cilkview`.

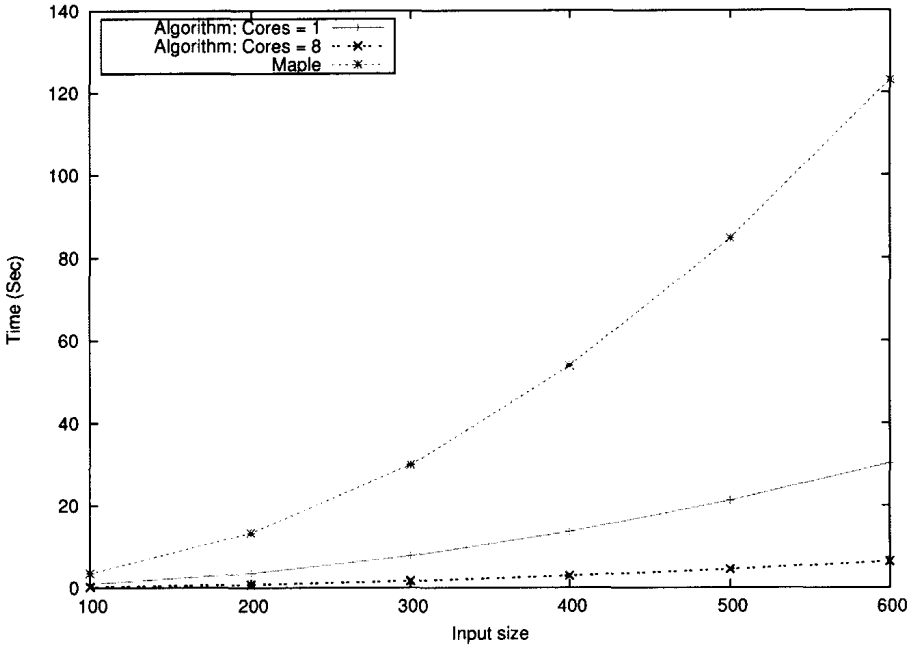

Figure 6.6: Running time comparisons of the augment refinement based parallel factor refinement algorithm for dense square-free univariate polynomials.

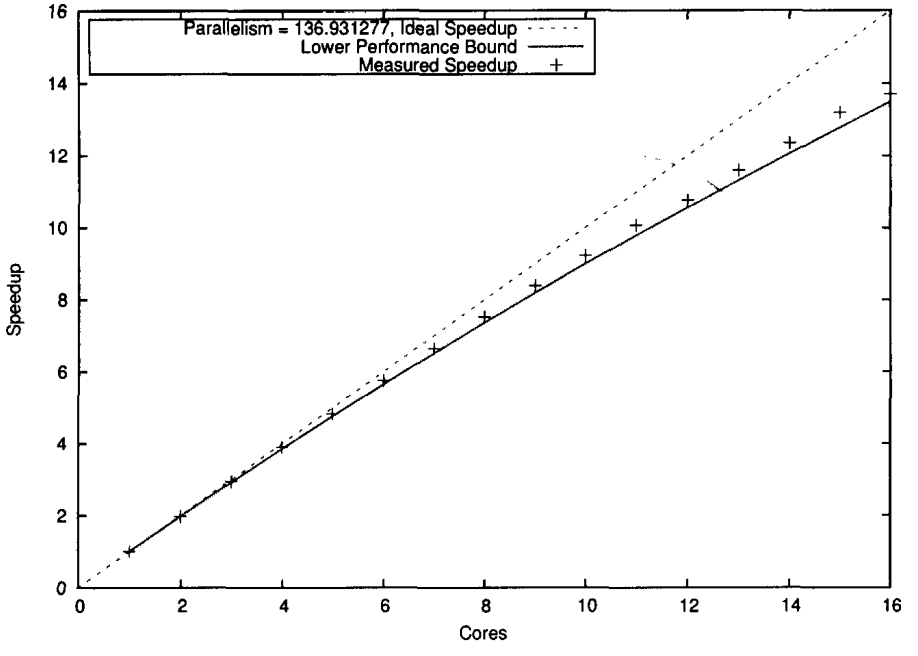

Figure 6.7: Scalability analysis of the augment refinement based parallel factor refinement algorithm for $4,120$ sparse square-free univariate polynomials by `Cilkview` when the input is already a GCD-free basis.

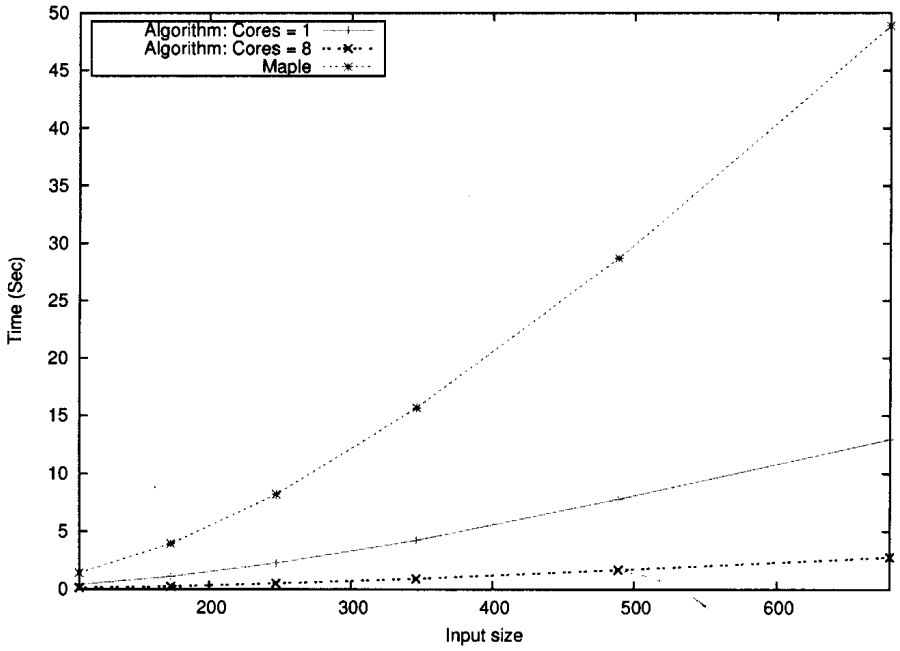

Figure 6.8: Running time comparisons of the augment refinement based parallel factor refinement algorithm for sparse square-free univariate polynomials when the input is already a GCD-free basis.

Chapter 7

Parallel GCD-free Basis Algorithm Based on Subproduct Tree Techniques

Parallel algorithms for computing coprime factorization for plain (or quadratic) arithmetic are discussed in Chapter 4. Fast arithmetic techniques are not applied there. In this chapter we will discuss the extension of asymptotically fast algorithms (Algorithm 11, 12, and 13) described in Section 3.3 to a parallel processing setting. These algorithms are designed based on the subproduct tree techniques presented in Section 2.3. The detailed description of these algorithms in a serial setting is available in [15] as well as in Section 3.3. We focus here on the parallel versions of these algorithms, leading to Algorithm 23, 24, and 25 where Algorithm 25 is the top level procedure. Section 7.1 is dedicated to these parallel algorithms and an analysis of their parallelism in the fork-join parallelism model, while in Section 7.3, we discuss challenges toward their implementation on multicore architectures.

7.1 Algorithms and parallelism estimates

Since Algorithms 11, 12, and 13 essentially rely on subproduct tree techniques, our first task is to parallelize the construction of subproduct trees, that is, Algorithm 2.

Observe that Algorithm 2 is stated for the case where the n nodes are univariate polynomials of degree 1. The same algorithm works correctly with polynomials of arbitrary degree. If the sum of their degree is d then Algorithm 2 runs within $O(M(d)\log_2(n))$ operations in $\mathbb{K}$. See Proposition 2 for the notations.

Obviously, parallelizing Algorithm 2 requires to parallelize univariate polynomial multiplication. In the fork-join parallelism model, a one-dimensional FFT computation of a vector of length d has a span of $O(\text{logp}(d)^2)$. Therefore, assuming that we are using an FFT-based univariate multiplication with $O(M(d) \in O(d\,\text{logp}(d))$ (as often in practice), we have the following result.

Proposition 20. *There exists a multithreaded algorithm computing the subproduct tree of n arbitrary polynomials $a_1, a_2, \ldots, a_n$ in $\mathbb{K}[x]$ with a work of $O(d\,\text{logp}(d)\ \log_2(n))$ and a span of $O(\text{logp}(d)^2 \log_2(n))$, where $d = \sum_{i=1}^{i=n} \deg(a_i)$.*

Our next task is to parallelize Algorithm 10. Since this algorithm involves GCD computations of univariate polynomials, we need to specify how we perform those.

Euclidean algorithms for univariate polynomial GCD computations in degree d run in $O(M(d)\text{logp}(d))$ operations in $\mathbb{K}$. In the fork-join parallelism model, they have a span of $O(d\,\text{logp}(d))$. Indeed, each division step runs in $O(\text{logp}(d))$. In the PRAM model, univariate polynomial GCDs can be computed in polylog time, but the corresponding algorithm has a work of $O(d^4)$ and is not suitable for a multicore implementation due to an overhead in memory consumption, This leads to us to the following simplification assumption.

Hypothesis 7. *For simplicity, we will assume that we have a multithreaded algorithm computing univariate polynomial GCD in degree d with a work of $O(M(d)\text{logp}(d))$ and a span of $O(d)$. This latter estimate can be established for systolic arrays, see [11].*

Proposition 21. *Assume that Algorithm 10 takes as input a polynomial f of degree less than d and n polynomials $a_1, \ldots, a_n$ with $d = \sum_{i=1}^{i=n} \deg(a_i)$. Then, in the fork-join parallelism model, Algorithm 10 can be executed with a work of $O(M(d)\text{logp}(d))$ and an expected span of $O(\log_2(n)\,\text{logp}(d)^2 + d/n)$.*

PROOF ▷ At Line 4, Algorithm 10 constructs a subproduct tree with a span of $O(\log_2(n)\,\text{logp}(d)^2)$. Then, at Line 5, it computes n GCDs concurrently, each of them in degree d/n on average. ◁

Algorithm 23 below is a parallel version of Algorithm 11. It takes two sequences of polynomials $A = a_1, a_2, \ldots, a_e$ and $B = b_1, b_2, \ldots, b_s$ as input and computes all pairs of GCDs $\gcd(a, b_j)$, for $1 \leq i \leq e$ and $1 \leq j \leq s$. This algorithm assumes that the polynomials in A (resp. B) are pairwise coprime.

Proposition 22. *Assume that Algorithm 23 takes as input two sequences of the same length $n = e = s$ and such that we have*

$$d = \sum_{i=1}^{i=n} \deg(a_i) = \sum_{i=1}^{i=n} \deg(b_i).$$

Then, Algorithm 23 has a span of $O(\frac{d}{n}\ \log_2(n))$.

PROOF ▷ Indeed, executing Algorithm 23 means traversing a binary tree, top-down level by level, such that this tree has $\log_2(n)$ levels each node has a span of $O(\log_2(n)\,\mathrm{logp}(d)^2 + d/n)$. ◁

Algorithm 23: `parallelPairsOfGcd`(A, B)

Input: Sequence of square-free polynomials $A = a_1, a_2, \ldots, a_e$ and $B = b_1, b_2, \ldots, b_s$ in $\mathbb{K}[x]$ such that the elements of A (resp. B) are pairwise coprime.

Output: $\gcd(a_1, b_1), \ldots, \gcd(a_1, b_s), \ldots, \gcd(a_e, b_1), \ldots, \gcd(a_e, b_s)$.

1: Build a subproduct tree called $Sub(a_1, a_2, \ldots, a_e)$ like Algorithm 2 where the root is labeled by the product of $a_1 a_2 \ldots a_e$ and let $f =$ `RootOf`(Sub);
2: Label the root of Sub by `multiGcd`(f, B) ;
3: **for** every node $N \in Sub$, going top-down, processing all nodes of the same level concurrently **do**
4: | **if** N is not a leaf and has label g **then**
5: | | $f_1 \leftarrow$ **spawn** `leftChild`(N);
6: | | $f_2 \leftarrow$ **spawn** `rightChild`(N);
7: | | **sync**;
8: | | Label f_1 by `multiGcd`(f_1, g);
9: | | Label f_2 by `multiGcd`(f_2, g);
10: Print the leaf labels in a in-fix traversal of the tree;

Algorithm 24 is a parallel version of Algorithm 12. It takes two sequences of square-free polynomials $A = a_1, a_2, \ldots, a_e$ and $B = b_1, b_2, \ldots, b_s$ as input and computes a GCD-free basis of A and B where the polynomials in A (resp. B) are pairwise coprime.

Proposition 23. *Assume that Algorithm 24 takes as input two sequences of the same length* $n = e = s$ *and such that we have*

$$d = \sum_{i=1}^{i=n} \deg(a_i) = \sum_{i=1}^{i=n} \deg(b_i).$$

Then, Algorithm 24 has an expected span of $O(\frac{d}{n}\ \log_2(n))$.

PROOF ▷ Indeed, executing Algorithm 24 requires

- executing Algorithm 23 at Line 1 with an expected span of $O(\frac{d}{n}\ \log_2(n))$,
- at Lines 4 and 8, computing subproduct trees in degree d/n (on average) with n items leading to an expected span of $O(\mathrm{logp}(d/n)^2 \log_2(n))$

- performing (fast) polynomial divisions in degree d/n (on average) at Lines 5 and 9, leading to an expected span of $O(\text{logp}(d/n)^3)$.

The conclusion follows. $\triangleleft$

Algorithm 24: `parallelGcdFreeBasisSpecialCase`(A, B)

Input: Sequence of polynomials $A = a_1, a_2, \ldots, a_e$ and $B = b_1, b_2, \ldots, b_s$ where, in each sequence, all polynomials are square-free and pairwise coprime.

Output: A sequence of polynomials forming a GCD-free basis of A, B.

1: $(g_{i,j})_{1 \leq i \leq e, 1 \leq j \leq s} \leftarrow$ `parallelPairsOfGcd`(A, B);
2: **parallel_for** $j = 1$ to s **do**
3: $\quad L_j \leftarrow$ `removeConstants`$(g_{1,j}, g_{2,j}, \ldots, g_{e,j})$; `// remove constant polynomials`
4: $\quad \beta_j \leftarrow \prod_{l \in L_j} l$;
5: $\quad \gamma_j \leftarrow b_j$ quotient β_j;
6: **parallel_for** $i = 1$ to e **do**
7: $\quad L_i \leftarrow$ `removeConstants`$(g_{i,1}, g_{i,2}, \ldots, g_{i,s})$; `// remove constant polynomials`
8: $\quad \alpha_i \leftarrow \prod_{l \in L_i} l$;
9: $\quad \delta_i \leftarrow a_i$ quotient α_i;
10: **return** `removeConstants`$(\{g_{1,1}, \ldots, g_{i,j}, \ldots, g_{e,s}, \gamma_1, \gamma_2, \ldots, \gamma_s, \delta_1, \delta_2, \ldots, \delta_e\})$; `// remove constant polynomials`

Algorithm 25 is a parallel version of Algorithm 13. It takes a sequence of non-constant square-free polynomials $A = a_1, a_2, \ldots, a_e$ as input and produces a GCD-free basis of A as output.

Theorem 3. *Assume that Algorithm 25 takes as input sequence A of n square-free polynomials such that we have*

$$d = \sum_{i=1}^{i=n} \deg(a_i).$$

Then, Algorithm 25 has a work of $O(M(d)\text{logp}(d)^3)$.

For an FFT-based parallel univariate multiplication with $M(d) \in O(d\,\text{logp}(d))$, Algorithm 25 has an expected span of

$$O(\frac{d}{n}\ \log_2(n)^2).$$

Consequently, for this multiplication, Algorithm 25 has an expected parallelism of

$$O(n\ \frac{\text{logp}(d)^4}{\log_2(n)^2}).$$

PROOF ▷ The work estimate was stated in Proposition 14. For the span, we observe that executing Algorithm 25 means traversing a binary tree, bottom-up level by level, such that each node at level i (for $i = 1, \ldots, \log_2(n)$) one needs to apply Algorithm 24 to 2^i polynomials with a degree sum of $2^i \frac{d}{n}$ (expectedly). Applying Proposition 23 this leads to a total expected span of $O(\frac{d}{n} \log_2(n)^2)$. ◁

Algorithm 25: `parallelGcdFreeBasis`(A)

Input: Sequence of square-free polynomials $A = a_1, a_2, \ldots, a_e$.
Output: A GCD-free basis of A.

1: Build a subproduct tree called $Sub'(A)$ like Algorithm 2 where the root is labelled by the sequence of polynomials A;
2: **for** every node $N \in Sub'$, bottom-up, processing all nodes of the same level concurrently **do**
3: | **if** N is not a leaf **then**
4: | | $f_1 \leftarrow$ **spawn** `leftChild`(N);
5: | | $f_2 \leftarrow$ **spawn** `rightChild`(N);
6: | | **sync**;
7: | | Label N by `parallelGcdFreeBasisSpecialCase`(f_1, f_2);
8: **return** the label of `RootOf`(Sub');

7.2 Asymptotic analysis of memory consumption

Hypothesis 8. *We assume that each element of the field $\mathbb{K}$ of each input or output sequence of polynomials (which themselves are assumed to be non-constant and monic) in Algorithms 10, 23, 24 or 25 can be stored in one machine word.*

Proposition 24. *Let d denote the sum of the degrees of the input polynomials in Algorithms 10, 23, 24 or 25, with input data of size n. Then, under Hypothesis 8, the space complexity of Algorithm 25 is $O(d \log(d))$ bits.*

PROOF ▷ Consider first Algorithm 23. Assuming that, at each tree level, each node is handled by a dedicated processor, then Algorithm 23 requires $O(\log(n)\, d)$ bits for storage. Assuming that the products at Lines 4 and 8 are performed by means of subproduct trees, Algorithm 24 requires $O(\log(d)\, d)$ bits for storage. Finally, using the superadditivity of the space storage estimates for Algorithm 24, we deduce that each tree level of Algorithm 25 can be processed within $O(\log(d)\, d)$ bits of storage. ◁

7.3 Challenges toward an implementation

Theorem 3 is a negative result. Indeed, the paralllelism estimate becomes $O(n)$ up to log factors. Therefore, no speed up comes from the parallelization of the polynomial multiplication. This is due to the fact that there is no practically efficient solution for parallelizing univariate polynomial GCD computation.

One could argue that an estimated paralllelism of $O(n)$ is not that bad. This is actually what we obtained with our algorithms based on quadratic polynomial arithmetic, see Proposition 18. In addition, this parallelism associated with a better work (thanks to asymptotically fast arithmetic) should lead to an efficient algorithm.

At this stage, one should take the targeted architecture into consideration. The works reported in [13] and [29] show that parallelizing fast algorithms for polynomial multiplication on multicore architecture is a hard problem. The parallelization overheads on this type of architecture make parallel implementation of these algorithms efficient from degree $100,000$ to $1,000,000$. This will have a very negative impact on the span of subproduct tree construction.

One should also observe that polynomial GCD computation are even harder to parallelize efficiently and our $O(d)$ span hypothesis is a very optimistic assumption. Finally, subproduct tree construction has also a negative impact on memory consumption and no cache-friendly algorithms is known to us for that task.

Chapter 8

Conclusion

Within Chapter 7 and Chapter 4, we have studied the parallelization of three algorithms for coprime factorization. For the one presented in Chapter 7 our theoretical results and pre-existing partial experimentation were not encouraging a multicore implementation. For the one presented in Section 4.1, its theoretical study was not discouraging in the first place and we decided to implement it. However, this did not bring satisfactory results. Finally, the parallel algorithm presented in Section 4.2 yielded very satisfactory results in practice.

It is interesting to compare the theoretical study of these latter two algorithms. First, we note that the cache complexity result of the algorithm of Section 4.1 is of the form $O(n^2/L)$ while the one of the algorithm of Section 4.2 is of the form $O(n^2/ZL)$. This implies that the ratio work to cache complexity is respectively L and ZL. The second ratio is prefered since the penalty for one cache miss in a multicore processor is typically in the order of 100 CPU cycles while a machine word operation may cost less than one CPU cycle, thanks to instruction level parallelism (ILP). One may wonder where did the Z factor disappear in the algorithm of Section 4.1. The trap is the work space used by Algorithm 14 (the a 2-D array G) which creates an overhead of cache misses. In fact, this array G is not really needed since most of its entries are just "1".

In conclusion, this research work shows that asymptotically fast algorithms do not always pay off on multicore architectures and that it is worth investing effort on algorithms based on plain arithmetic. In addition, this research work shows that the *à la FORTRAN* "work space passed as parameter" can create a performance bottleneck in terms of cache complexity. Moreover, it is worth investing effort on sophisticated divide-and-conquer algorithms, such as Algorithm 22, which are able to save on memory consumption and reduce memory traffic.

Bibliography

[1] GMP: Arithmetic without limitations. `http://gmplib.org`.

[2] Maple. `http://www.maplesoft.com`.

[3] Sharcnet: Computing tomorrow's solutions. `https://www.sharcnet.ca`.

[4] L. A. Belady. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal*, 5:78–101, 1966.

[5] P. Aubry, D. Lazard, and M. Moreno Maza. On the Theories of Triangular Sets. *J. Symb. Comput.*, 28:105–124, July 1999.

[6] E. Bach, J. Driscoll, and J. Shallit. Factor Refinement. In *Symposium on Discrete Algorithms*, pages 201–211, 1990.

[7] L. Bernardin. On Square-free Factorization of Multivariate Polynomials over a Finite Field. *Theoretical Computer Science*, 187:105–116, November 1997.

[8] D. J. Bernstein. Factoring into Coprimes in Essentially Linear Time. *J. Algorithms*, 54(1):1–30, 2005.

[9] D. J. Bernstein, H. W. Lenstra Jr., and J. Pila. Detecting Perfect Powers by Factoring into Coprimes. *Math. Comput.*, 76(257):385–388, 2007.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:356 – 368, 1994.

[11] R. P. Brent, H. T. Kung, and F. T. Luk. Some Linear-time Algorithms for Systolic Arrays. In *IFIP Congress*, pages 865–876, 1983.

[12] A. Brown, editor. *VLSI Circuits and Systems in Silicon.* McGraw-Hill, Inc., New York, NY, USA, 1991.

[13] M. F. I. Chowdhury, M. Moreno Maza, W. Pan, and É. Schost. Complexity and Performance Results for non FFT-based Univariate Polynomial Multiplication, 2011.

[14] Intel corporation. Cilk++. `http://www.cilk.com`.

[15] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. On the Complexity of the D5 Principle. *SIGSAM Bull.*, 39:97–98, September 2005.

[16] J. Dennis. On Newton's Method and Nonlinear Simultaneous Replacements. *SIAM J. Numer. Anal.*, 4:103 – 108, 1967.

[17] J. Dennis. On Newton-like Methods. *Numerische Mathematik*, 11:324 – 330, 1968. 10.1007/BF02166685.

[18] G. E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition–Preliminary Report. *SIGSAM Bull.*, 8:80–90, August 1974.

[19] C. E. Leiserson, L. Li, M. Moreno Maza, and Y. Xie. Parallel Computation of the Minimal Elements of a Poset. In *4th International Workshop on Parallel and Symbolic Computation*, pages 53–62, 2010.

[20] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and Other Cilk++ Hyperobjects. In *SPAA '09: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, pages 79–90, New York, NY, USA, 2009. ACM.

[21] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–297, New York, USA, October 1999.

[22] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN*, 1998.

[23] M. Frigo and V. Strumpen. The Cache Complexity of Multithreaded Cache Oblivious Algorithms. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 271–280, New York, NY, USA, 2006. ACM.

[24] J. Hong and H. T. Kung. I/O Complexity: The Red-blue Pebble Game. In *STOC*, pages 326–333, 1981.

[25] M. Kalkbrener. *Three Contributions to Elimination Theory.* PhD thesis, Johannes Kepler University, Linz, 1991.

[26] C. E. Leiserson. The Cilk++ Concurrency Platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA, 2009. ACM.

[27] Y. Lu. Artificial Intelligence in Mathematics. chapter Searching dependency between algebraic equations: an algorithm applied to automated reasoning, pages 147–156. Oxford University Press, Inc., New York, NY, USA, 1994.

[28] J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables.* Academic Press (New York), 1970.

[29] M. Moreno Maza and Y. Xie. Balanced Dense Polynomial Multiplication on Multi-cores. *Int. J. Found. Comput. Sci.*, 22(5):1035–1055, 2011.

[30] M. Moreno Maza and Y. Xie. FFT-based Dense Polynomial Arithmetic on Multi-cores. In D.J.K. Mewhort, editor, *Proc. HPCS 2009*, volume 5976 of *LNCS*, Heidelberg, 2010. Springer-Verlag Berlin.

[31] P. S. Wang. The EEZ-GCD Algorithm. *SIGSAM Bull.*, 14:50–60, May 1980.

[32] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[33] J. von zur Gathen. Representations and Parallel Computations for Rational Functions. *SIAM J. Comput.*, 15:432–452, May 1986.

[34] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[35] D. Y. Y. Yun. On Square-free Decomposition Algorithms. In *Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation*, SYMSAC '76, pages 26–35, New York, NY, USA, 1976. ACM.